PORTFOLIO

JORGE VIÑUELA PÉREZ

PROJECTS

Engineering Cycle, Mixed Signal Processing and PCB Design on the Development of an Electronic Dice

Task 1: Define the problem statement

This device should meet the following key objectives:

- **Compact size:** No larger than 70mm x 50mm.
- **Multi-dice Functionality:** Dice types d4, d6, d8, d10, d12 and d20.
- **Visual display:** Reveals the result of each roll.
- **True randomness**
- **User-friendly interface:** A button to switch between dice types.
- **Extended battery life**

Based on the objectives, here's a Gantt Chart for the estimated timeline of the project.

Activity	Week 1	Week 2	Week 3
Define the problem statement			
Research and analysis			
Brainstorming			
Concept Development – Hardware			
Concept Development – Software			
Feasibility Assessment			
Detailed Design and Planning – Schematic Design			
Detailed Design and Planning – PCB Design			
Detailed Design and Planning – Material and Resource Planning			
Field Ready Prototype and Testing and Evaluation			
Optimization and Improvement			
Design a 3D model case for the product			

Table 1. Electronic Dice Project's Gantt Chart

Task 2: Research and analysis

For this task, I focused on identifying the components needed to design and build the product. I researched existing products and suppliers to find parts that met the project requirements, considering cost, availability, and compatibility. The goal was to ensure the components are efficient, compact, and aligned with the client's needs.

Component Research and Justification

The table below summarizes the key components I considered initially and the rationale behind their selection:

Supplier and Cost Overview

I sourced the components from approved suppliers like Mouser Electronics, Digi-Key, and Model Rockets to ensure high quality and reliability. The table below shows the details of the selected components, including cost, quantity, and manufacturer. Here are key observations from the initial list of components needed.

Product Nam	Product Desc	URL	Distributor	Manufacturer	Cost (inv VAT)	Quantity	Total
ATMEGA328P	8-bit Microcor	https://www.r	Mouser Electr	Microchip	£2.10	1	£2.10
7-segment (HI	LED Displays &	https://www.r	Mouser Electr	Broadcom Lim	£1.55	2	£3.10
Tactile Switch	Tactile Switch	https://www.r	Mouser Electr	Apem	£0.32	2	£0.64
Usb B Micro (U	USB Connecto	https://www.r	Mouser Electr	GCT	£0.56	1	£0.56
Lipo Battery (M	3.7V 150mAh	https://modell	Model Rockets	PKCell	£8.75	1	£8.75
Step-Up Voltag	Switching Volt	https://www.r	Mouser Electr	Texas Instrum	£3.02	1	£3.02

Table 2. Initially Needed Components List

1. Total Cost

   The total cost of the main components is approximately £18, including VAT but not including shipping, which fits within a reasonable budget for prototyping.

2. Suppliers and Reliability

   Most components were sourced from Mouser Electronics, with the battery sourced from Model Rockets. All suppliers are reputable, ensuring reliable components and availability.

3. Compatibility

   All components are compatible with the 3.3V–5V operating range of the system. For example, the LiPo battery and step-up voltage regulator ensure that the microcontroller and display receive stable power.

Observations and Challenges

- Component Size: The compact size of components like the ATMEGA328P-AU and the 7-segment display ensures that the project can meet the size constraint.
- Power Efficiency: I selected a LiPo battery from Model Rockets and a voltage regulator to optimize the dice for extended use, ensuring low power consumption, while the system is portable.
- Cost vs. Quality: While some cheaper alternatives were available, I prioritized components with a proven track record to avoid issues during prototyping.

Task 3: Brainstorming

For the microcontroller, there were a few options on the table. Initially, we thought about using an Arduino UNO, given its familiarity and ease of use. However, its size was a

drawback for this project. We also considered the ATtiny85, which is compact but has only 3 pins, limiting the number of components it could support. The ATtiny84 offered more pins (12), but we felt it still might be too restrictive for the design's complexity. Finally, we chose the ATmega328P, a small and compact device, yet it offers enough I/O pins to handle all the necessary components without compromising functionality.

Next, we discussed the power supply. Since the device needs to be portable, we quickly dismissed the idea of using a standard 9V battery, which would be bulky and require frequent replacements. Instead, we opted for a rechargeable LiPo battery. To make recharging easy, we decided to integrate a USB-B micro connector for charging, ensuring that the device could be conveniently powered up without needing external chargers or frequent battery replacements.

For the display, the main challenge was finding something that would clearly show the numbers without consuming too much power or requiring too many pins. Initially, we investigated using an OLED or e-ink display, but these would require more complex programming and higher power consumption than necessary. We settled on two 7-segment displays because we could use multiplexing, which would allow us to control both displays with fewer pins, further optimizing the design.

Draw a high-level flow chart for the product.

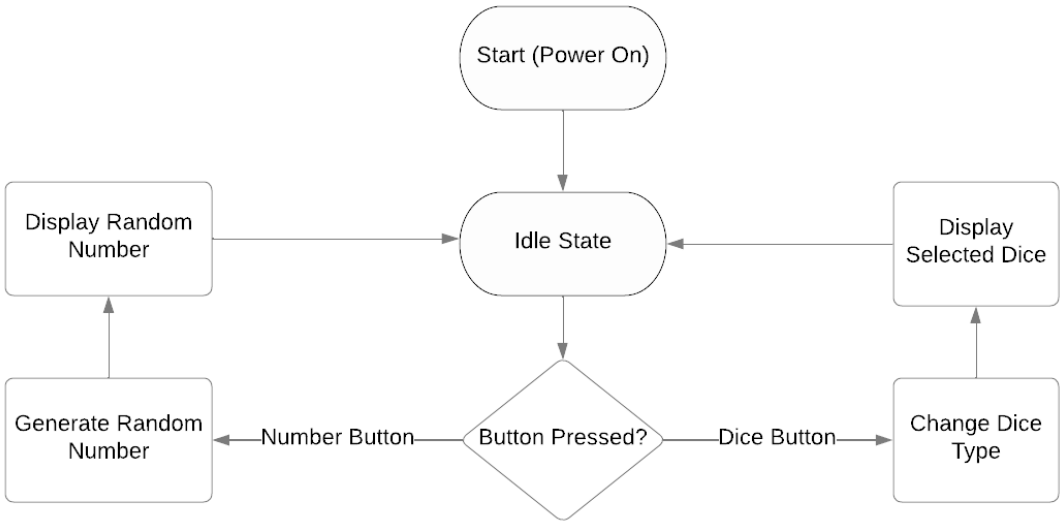


Figure 1. Electronic Dice Flow Chart

Draw a logic diagram for the software of the product.

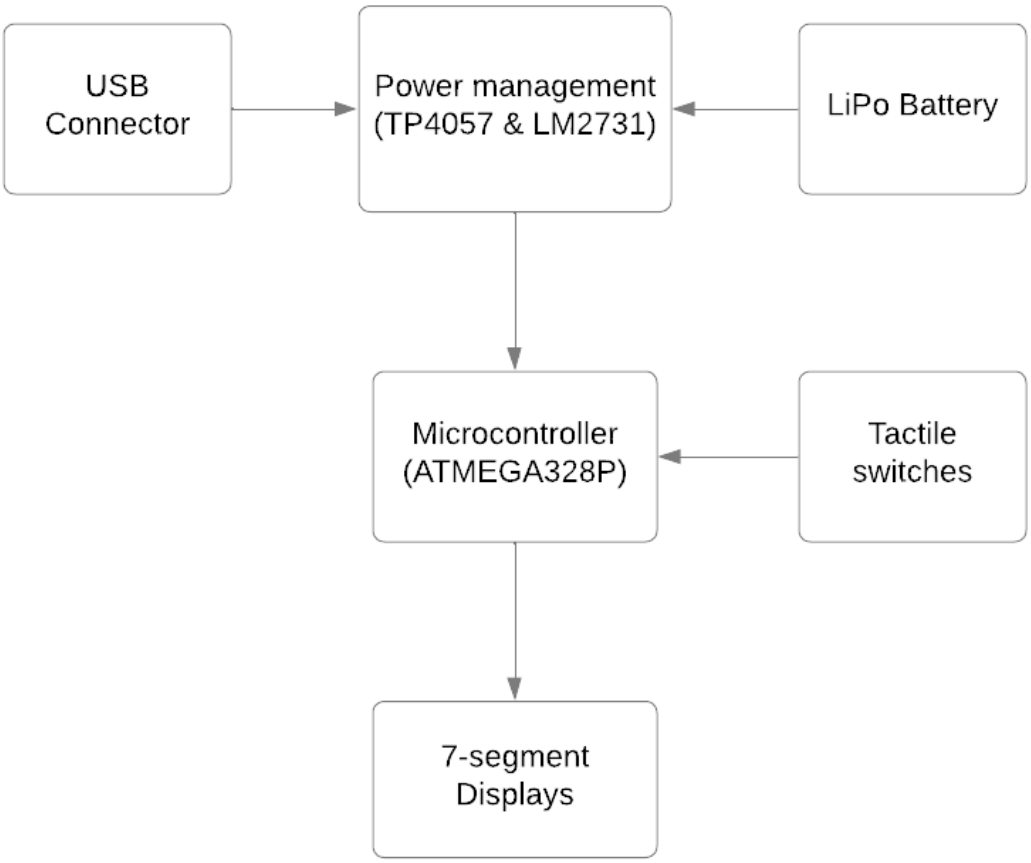


Figure 2. Electronic Dice Logic Diagram

Task 4: Concept Development – Hardware

The initial hardware prototype for the project involves using an Arduino UNO, which incorporates the ATmega328P microcontroller, along with two buttons and two 7-segment displays to visualize the results. The system simulates a dice rolling mechanism with multiple dice types (d4, d6, d8, d10, d12, and d20). Here's a breakdown of how the hardware is connected:

- **Buttons**: The first button is used to switch between different dice types (d4, d6, etc.). Each press cycles through the different dice options and shows on the display the selected dice. The second button is responsible for generating a random number within the selected dice range.
- **7-Segment Displays**: The output is visualized on two 7-segment displays, which are used to show the dice result. The segments (A-G) of both 7-segment displays are wired together to the same GPIO pins of the ATmega328P.

The cathodes of the two displays are connected to separate GPIO pins on the microcontroller. This allows for multiplexing, where each display can be controlled individually by activating its corresponding cathode while turning off the other display. This way, we can display different numbers on each display despite sharing the segment connections.

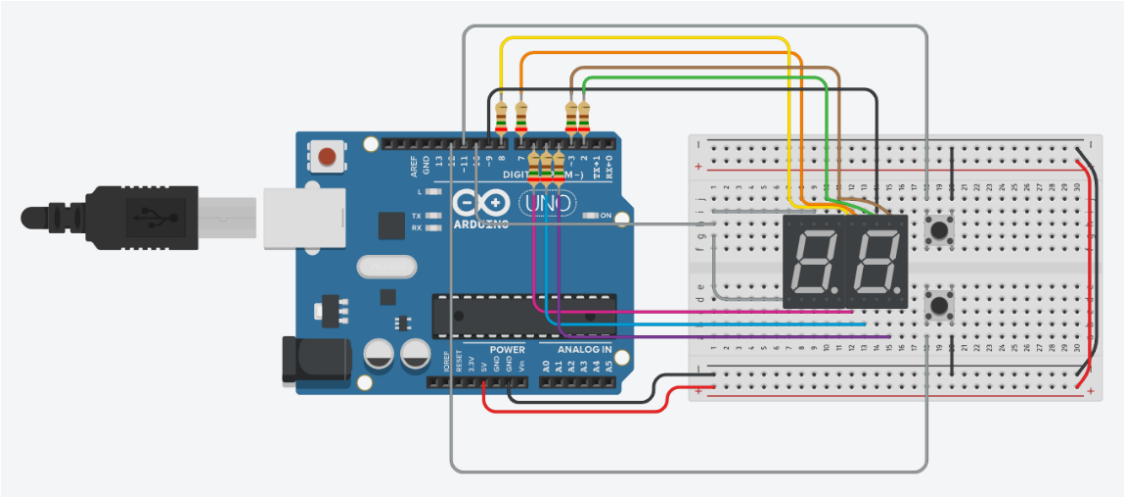


Figure 3. Electronic Dice Hardware Concept

Task 5: Concept Development – Software

Once the main hardware concept was outlined, including the microcontroller, buttons, and displays, the next step was to integrate the corresponding software. The software is tightly coupled with the hardware, ensuring that all components work seamlessly together to achieve the intended functionality. The code controls the dice rolling system,

manages user inputs, and displays the results in real time on two 7-segment displays. Here's a breakdown of the code and how it functions:

1. Pin Assignments:
   - Pins pinA through pinG control the segments of the 7-segment display (common across both displays).
   - display1Pin and display2Pin are used for multiplexing between the two displays, allowing each digit to be shown one at a time.
   - buttonDiceType is for selecting the type of dice, and buttonRoll triggers the roll to generate a new random number.

Dado.ino

```
1   // Pins for the segments of the display
2   const int pinA = 2;
3   const int pinB = 3;
4   const int pinC = 4;
5   const int pinD = 5;
6   const int pinE = 6;
7   const int pinF = 7;
8   const int pinG = 8;
9
10  const int display1Pin = 9;  // First display
11  const int display2Pin = 10; // Second display
12
13  const int buttonDiceType = 11; // Type of dice
14  const int buttonRoll = 12;     // Random number
15
16  int diceType = 6; // Initial dice: d6
17  int number = 1;   // Generated Number
18  int displayValue = 6;
```

Figure 4. Electronic Dice Initialization

2. Initial Variables and Constants:
   - diceType starts at 6, representing a d6 dice by default.
   - displayValue stores the number currently displayed on the displays.
   - digitSegments array holds the binary encoding for each digit to control which segments are turned on for each number.

3. Setup Function:
   - Configures segment pins and display control pins as outputs.
   - Sets up buttons with INPUT_PULLUP mode, enabling internal pull-up resistors.

```
// Configuration of each digit
const byte digitSegments[10] = {
  0b0111111, // 0
  0b0000110, // 1
  0b1011011, // 2
  0b1001111, // 3
  0b1100110, // 4
  0b1101101, // 5
  0b1111101, // 6
  0b0000111, // 7
  0b1111111, // 8
  0b1101111  // 9
};

void setup() {
  // Pins of the segments as outputs
  pinMode(pinA, OUTPUT);
  pinMode(pinB, OUTPUT);
  pinMode(pinC, OUTPUT);
  pinMode(pinD, OUTPUT);
  pinMode(pinE, OUTPUT);
  pinMode(pinF, OUTPUT);
  pinMode(pinG, OUTPUT);

  // Control pins as outputs
  pinMode(display1Pin, OUTPUT);
  pinMode(display2Pin, OUTPUT);

  // Button pins as Inputs
  pinMode(buttonDiceType, INPUT_PULLUP);
  pinMode(buttonRoll, INPUT_PULLUP);
}
```

Figure 5. Electronic Dice Setup Function

4. Loop Function:
   - When buttonDiceType is pressed, it calls changeDiceType function to cycle through different dice types (d4, d6, d8, d10, d12, and d20).
   - When buttonRoll is pressed, it calls rollDice function to generate a random number within the current dice range.
   - After handling button inputs, the displayNumber(displayValue); function is called to show the current number on the 7-segment displays with multiplexing.

5. Dice Type Switching (changeDiceType):
   - It cycles through the dice types in this order: d4 → d6 → d8 → d10 → d12 → d20, then back to d4.

```
53  void loop() {
54
55    if (digitalRead(buttonDiceType) == LOW) {
56      delay(200);
57      changeDiceType();
58    }
59
60    if (digitalRead(buttonRoll) == LOW) {
61      delay(200);
62      rollDice();
63    }
64
65    // Constant multiplexing to display the value on both displays
66    displayNumber(displayValue);
67  }
68
69  void changeDiceType() {
70    switch (diceType) {
71      case 4: diceType = 6; displayValue = 6; break;
72      case 6: diceType = 8; displayValue = 8; break;
73      case 8: diceType = 10; displayValue = 10; break;
74      case 10: diceType = 12; displayValue = 12; break;
75      case 12: diceType = 20; displayValue = 20; break;
76      case 20: diceType = 4; displayValue = 4; break;
77    }
78  }
```

Figure 6. Electronic Dice Loop Function

6. Dice Rolling (rollDice):
    - It generates a random number between 1 and the maximum value of the selected dice type and stores it on the DisplayValue variable.

7. Displaying the Number (displayNumber):
    - This function splits the number into tens and units.
    - Each digit is displayed on its respective display using multiplexing, with a brief delay to switch between digits smoothly.

8. Setting Segments (setSegments):
    - This function uses bitwise operations to activate the correct segments for a given digit by controlling each segment pin.

```
80  void rollDice() {
81    number = random(1, diceType + 1);
82    displayValue = number;
83  }
84
85  void displayNumber(int num) {
86    int tens = num / 10;
87    int units = num % 10;
88
89    // Show first digit
90    digitalWrite(display1Pin, HIGH);
91    digitalWrite(display2Pin, LOW);
92    setSegments(digitSegments[tens]);
93    delay(40); // Delay for the multiplexing
94
95    // Show second digit
96    digitalWrite(display1Pin, LOW);
97    digitalWrite(display2Pin, HIGH);
98    setSegments(digitSegments[units]);
99    delay(40);
100 }
101
102 // Turn on segments depending on the case
103 void setSegments(byte segments) {
104   digitalWrite(pinA, segments & 0b0000001);
105   digitalWrite(pinB, segments & 0b0000010);
106   digitalWrite(pinC, segments & 0b0000100);
107   digitalWrite(pinD, segments & 0b0001000);
108   digitalWrite(pinE, segments & 0b0010000);
109   digitalWrite(pinF, segments & 0b0100000);
110   digitalWrite(pinG, segments & 0b1000000);
111 }
```

Figure 7. Electronic Dice RollDice Function

Task 6: Feasibility Assessment

In this task, I evaluated the technical, economic, and operational feasibility of the product design, by analysing the finalized component selection, costs, and compatibility. This way, I ensured that the design aligns with the project's constraints and objectives.

Below is the finalized table of components, including their descriptions, suppliers, costs, and quantities:

Product Name	Product Description	URL	Distributor	Manufacturer	Cost (inv VAT)	Quantity	Total
ATMEGA328P-AU	8-bit Microcontroller:	https://www.r	Mouser Electronics	Microchip	£2.10	1	£2.10
7-segment (HDSP-513A)	LED Displays & Acce:	https://www.r	Mouser Electronics	Broadcom Limited	£1.55	2	£3.10
250R Resistor (OK15G5E-R52)	Carbon Film Resistor	https://www.r	Mouser Electronics	Ohmite	£0.09	7	£0.63
Tactile Switch (PHAP3344R)	Tactile Switches TAC	https://www.r	Mouser Electronics	Apem	£0.32	2	£0.64
TP4057	SOT23-6 BATTERY M/	https://www.c	DigiKey	UMW	£0.42	1	£0.42
Usb B Micro (USB3080-30-00-A	USB Connectors Mic	https://www.r	Mouser Electronics	GCT	£0.56	1	£0.56
JST-PH 2 Pin Connector (B2B-P	CONN HEADER VERT	https://digikey	DigiKey	JST Sales America Inc.	£0.16	1	£0.16
Led Charging (TLPR5600)	LED RED DIFF SIDE VI	https://www.c	DigiKey	Vishay Semiconductor Optc	£0.60	1	£0.60
Led Charged (TLPG5600)	LED GREEN DIFF SIDI	https://www.c	DigiKey	Vishay Semiconductor Optc	£0.60	1	£0.60
Ceramic Capacitor (CL05B103J	CAP CER 0.1UF 50V >	https://www.c	DigiKey	Samsung Electro-Mechanic	£0.10	2	£0.20
Electrolitic Capacitor (8650806	CAP ALUM 10UF 20%	https://www.c	DigiKey	Würth Elektronik	£0.18	2	£0.36
1k Resistor (CF1/2CT52R102J)	RES 1K OHM 5% 1/2V	https://www.c	DigiKey	KOA Speer Electronics, Inc.	£0.11	2	£0.22
2k Resistor (CF1/4CT52R202J)	RES 2K OHM 5% 1/4V	https://www.c	DigiKey	KOA Speer Electronics, Inc.	£0.09	2	£0.18
Lipo Battery (M105318)	3.7V 150mAh LiPo ba	https://modeli	Model Rockets	PKCell	£8.75	1	£8.75
Step-Up Voltage Regulator (LM2	Switching Voltage Re	https://www.r	Mouser Electronics	Texas Instruments	£3.02	1	£3.02
1.2k Resistor (CFM12JT1K20)	RES 1.2K OHM 5% 1/	https://www.c	DigiKey	Stackpole Electronics Inc	£0.08	1	£0.08
3.9k Resistor (CF14JT3K90)	RES 3.9K OHM 5% 1/	https://www.c	DigiKey	Stackpole Electronics Inc	£0.08	1	£0.08
SMD Power inductor (SRR1210-	Power Inductors - SM	https://www.r	Mouser Electronics	Bourns	£0.96	1	£0.96
10k Resistor (MCT0603MD1052	Thin Film Resistors - S	https://www.r	Mouser Electronics	Vishay / Beyschlag	£0.34	2	£0.68
						Total	**£23.34**

Table 3. Final List of Components

Technical Feasibility

The selected components meet the design's technical requirements and are compatible with each other to achieve the functionality of the product. Below is a detailed analysis of how each component contributes to the system:

1. Core Components
    - **Microcontroller (ATMEGA328P-AU):** It provides sufficient GPIOs to control the two displays and reads inputs from the tactile buttons, while it supports low-power operation.
    - **7-segment displays (HDSP-513A)**: The use of two of these components creates a compact two-digit display for showing dice roll results clearly. Efficient power consumption and compatible with the regulated 5V supply.

2. Input Components
    - **Tactile Switches (PHA3344R)**: Two buttons necessary, one for cycling through dice types and one for generating the random number. I selected Single Pole Single Throw (SPST) switches, which ensure reliable user input with robust tactile feedback.

3. Power Management Components
    - **LiPo Battery (3.7V, 150mAh)**: Compact and lightweight battery chosen mainly to meet the device's portability requirements, while providing sufficient capacity for prolonged usage, powering the microcontroller, displays and peripheral components. Additionally, the selected battery includes a built-in security circuit, ensuring it never truly powers off, thus protecting the device from abrupt power loss and preserving system stability during usage.

- **Step-Up Voltage Regulator (LM2731YMF)**: Boosts the 3.7V output of the battery to 5V, ensuring stable voltage for all the components in the product, while it efficiently regulates power to avoid unnecessary losses and extending battery life.
- **Battery Charging IC (TP4057)**: Allows recharging the LiPo battery via a micro-USB connector, ensuring safe charging with built-in overcharge protection.
- **Capacitors**
    - **Ceramic Capacitors (0.1µF)**: Used for decoupling to stabilize voltage and filter noise in the power supply, ensuring smooth operation of the microcontroller and display.
    - **Electrolytic Capacitors (10µF)**: Provide additional power stability, particularly for the voltage regulator during periods of higher current draw.

4. Passive Components
    - **250Ω Resistors**: Current-limiting resistors for the 7-segment displays, ensuring that LEDs are not overdriven.
    - **1kΩ and 3.9kΩ Resistors**: Pulldown or pullup resistors for tactile switches to ensure stable signal readings from the buttons.

5. Connectivity Components
    - **Micro-USB Connector**: Robust and compact design that allows the device to be charged conveniently using standard USB power sources,
    - **JST Connector**: Provides secure and compact connections for the battery, making assembly and maintenance easier.

6. Additional Considerations
    - **Multiplexing for 7-segment Displays**: By multiplexing (the microcontroller alternates between the tens and units digits quickly enough to make it appear as though both digits are displayed at the same time) the two displays, I successfully reduced the number of GPIO pins needed to control the segments.

    - **Size Constraints**: The use of compact components such us the microcontroller or the battery fit in the key-fob-sized requirements. Additionally, SMD versions of passive components further minimize the overall PCB size.

The selected components are expected to operate seamlessly together under the specified voltage conditions (5V, which preventing flickering or glitches). With this efficient power management, the device is projected to offer extended usage on single charge while maintaining functionality.

Economic Feasibility

This assessment is evaluated based on the total cost of components, sourcing reliability, production scalability, and the cost-benefit considerations of design decisions. Below is an in-depth analysis of the cost and financial implications of the project:

1. Component Cost Analysis
    - The total cost of components, including VAT, is £22.63 for a single unit. This cost is reasonable for a prototyping phase where parts are sourced individually and not in bulk (Shipping not included).

2. Bulk Production Potential
    - **Savings in Bulk Orders**: Many components can be purchased at significantly reduced prices when ordered in bulk, like passive components or microcontrollers. A rough estimate shows that bulk purchasing could decrease the component cost per unit by 20-30%, bringing the cost closer to £16 per unit in a hypothetical larger scale production.
    - **Scalability**: By transitioning to Surface Mount Devices (SMD) for passive components, assembly time and labour costs can be reduced in large scale manufacturing. A single PCB design can accommodate all components for automated assembly, which lowers production costs.
    - **Value vs. Cost:** In the prototype stage, £22.63 per unit is justified by the need for high-quality components to ensure functionality during testing. Moreover, the ability to recharge the device adds value to the design, avoiding the recurring cost of disposable batteries. On the other hand, for a retail product it would be essential to find cost reduction strategies to achieve a profitable margin while keeping the final product affordable.

3. Challenges
    - **Initial Cost for Prototyping**: Prototyping often incurs higher costs due to the lack of economies of scale.
    - **Logistics**: The need to source the components from multiple suppliers increases the shipping costs and complexity. Consolidating suppliers or sourcing locally could benefit the overall cost

4. Future Cost Optimization
    - **Component Consolidation**: Explore integrated IC solutions that combine microcontroller, power management, and charging capabilities to reduce the total component count and cost.
    - **Supplier Negotiations**: Negotiate with suppliers for better pricing when ordering in bulk.

The project is economically feasible for prototyping and offers significant potential for cost reduction in bulk production. By balancing quality, functionality, and affordability, the design ensures a competitive product with a strong value proposition. Future cost optimization strategies will further enhance its viability for commercial production.

Operational Feasibility

The operational feasibility of the product focuses on the ease of assembly, user functionality, portability, and maintainability. Below is a detailed analysis of how the design meets these operational objectives:

1. Ease of Assembly
    - **Component Placement:** The use of through-hole components, like the tactile switches and resistors, simplifies prototyping and assembly during the initial design stage. However, for larger scale production it would be necessary to transition to SMD components. A well-optimized PCB design ensures that components are logically and efficiently placed, minimizing assembly errors.
    - **Modular Design:** Critical parts, such as the battery or the USB connector, are easily replaceable, which ensures straightforward repairs or upgrades if needed. Additionally, the use of a JST connector allows for simple disconnection and reconnection without soldering, if needed.

2. User Functionality
    - **Dice Rolling Process:** The design provides a simple and intuitive interface by using only two buttons, one for selecting the dice and the other for rolling it. The user sees the both the selected dice and the generated number on the 7-segments displays, with multiplexing ensuring smooth digit transitions.
    - **Multi-Dice Compatibility:** The device supports multiple dice types, as required by the client, which increases the utility and appeal of the device.
    - **Portability:** The compact design ensured the dice is lightweight and easy to carry and the use of a rechargeable LiPo battery eliminates the need for

disposable ones, making it more user-friendly and environmentally sustainable.

- **Display Clarity:** The two displays provide bright and readable numbers even in low-light conditions.

3. Power Efficiency
    - **Battery Life:** The 3.7V 150mAh LiPo battery provides sufficient capacity for extended use, especially with the low-power ATMEGA328P-AU microcontroller. Efficient power management is achieved using the step-up voltage regulator to provide a stable 5V output, ensuring all components operate optimally without wasting energy.
    - **Charging Process:** The TP4057 charging IC allows the device to be conveniently recharged via a micro-USB port, ensuring ease of use and minimizing downtime.

4. Maintainability
    - **Repairability:** Key components like the LiPo battery and tactile switches are modular and can be easily replaced in case of failure.
    - **Software Updates:** The ATMEGA328P-AU can be reprogrammed via its ISP (In-System Programming) interface, enabling software updates or bug fixes without hardware changes.
    - **Component Durability:** High-quality components ensure long-term reliability.
5. Challenges and Risks
    - **Assembly Precision:** Implementing multiplexing requires precise wiring and logic to ensure functionality, given that errors during assembly could lead to display glitches.
    - **Power Optimization:** While the device is designed to efficient, real-world testing is required to validate the expected battery life under usage patterns.
    - **Size Constraints:** Assembling all components into the size requirements while maintaining functionality and durability require careful PCB design and component placement.

The design is operationally feasible, balancing ease of use, assembly simplicity, and portability. While challenges such as size constraints and power efficiency require testing and optimization, the design meets the project's requirements and is well-suited for prototyping and eventual production.

Task 7: Detailed Design and Planning – Schematic Design

The electronic dice circuit is designed to generate random dice rolls and display the results on a dual 7-segment LED display. The design is divided into two main phases:

1. **Power Management Phase:** This phase handles the power input, battery charging, and voltage regulation to ensure a stable 5V supply for the circuit. It includes a micro-USB input, TP4057 battery charging IC, and LM2731 step-up voltage regulator.
2. **Control and Display Phase**: This phase consists of the ATMEGA328P microcontroller, tactile switches, and two 7-segment displays. The microcontroller handles user inputs from the switches to select dice types and roll numbers. It also manages the multiplexing and control of the 7-segment displays, ensuring the rolled number is clearly shown to the user.

Power Management Phase

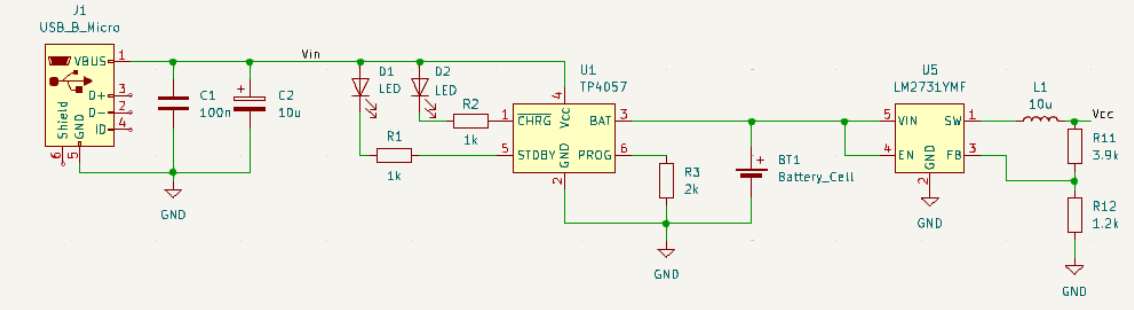


Figure 8. Power Management Phase Electric Scheme

In this section, the connections of each component in this phase will be explained in detail.

1. <u>Power Input (USB Connector)</u>
   - **Component: J1 (USB_B_Micro)**:
     - This connector provides power input to the system via a micro-USB port.
     - Pin 1 (VBUS): The power pin receives 5V from a standard USB source.
     - Pin 5 (GND): The ground connection.
   - **Capacitors:**
     - These are decoupling capacitors placed close to the USB connector to stabilize the input voltage and filter out noise.

- C1 (100nF) handles high-frequency noise, while C2 (10µF) smooths out low-frequency fluctuations.

2. Battery Charger TP4057 and Battery Cell
   - **Charger IC: TP4057 (U1)**:
     - The TP4057 is a lithium-ion battery charging IC that ensures safe and efficient charging of the connected battery cell.
     - Pin 1 (CHRG): Connects to LED D1, which indicates charging status.
     - Pin 2 (STDBY): Connects to LED D2, which indicates when the charging is complete.
     - Pin 3 (PROG): Connected to resistor R3 (2kΩ), which sets the charging current, of approximately 500mA.
     - Pin 5 (BAT): Connected to the positive terminal of the battery.
     - Pin 6 (VIN): Receives the stabilized 5V from the USB input (Vin).
     - Pin 4 (GND): Ground connection for the charger circuit.
   - **Battery Cell (BT1):** A 3.7V lithium-ion battery is connected at this point, charged by the TP4057.

3. Voltage Regulator
   - **Voltage Regulator IC: LM2731YMF (U5)**:
     - This step-up voltage regulator increases the battery's 3.7V to a stable 5V output for the rest of the circuit.
     - Pin 5 (VIN): Accepts input voltage from the battery.
     - Pin 1 (EN): Enable pin is tied to VIN, ensuring the regulator is always on when the battery is connected.
     - Pin 3 (FB): Feedback pin connected to a resistor divider (R11 and R12) to set the output voltage, of 5V.
     - Pin 2 (SW): The switching output is connected to an inductor (L1) to boost the voltage.

Control and Display Phase

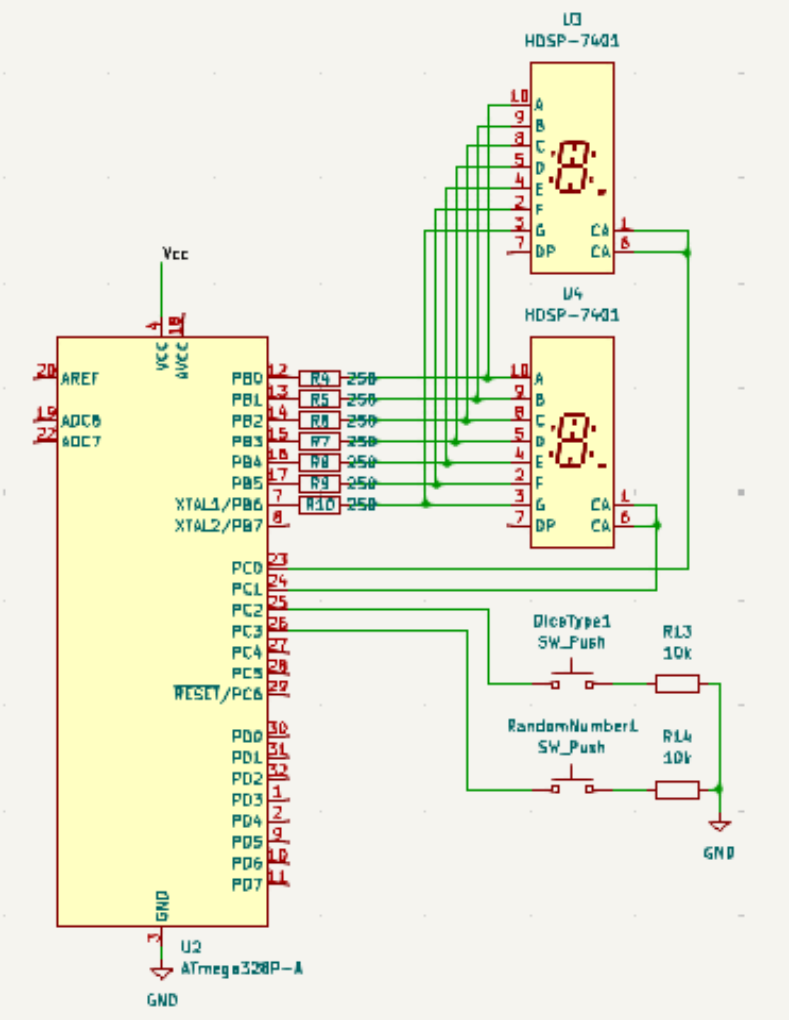


Figure 9. Control and Display Electric Scheme

In this section, the connections of each component in this phase will be explained in detail.

1. Microcontroller: ATMEGA328P (U2)
   - **Power Supply**:

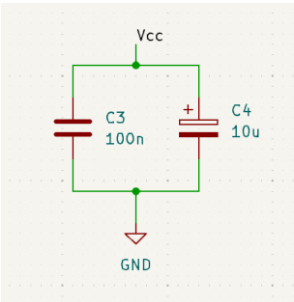


Figure 10. ATMEGA328P Power Supply

   - Connected to Vcc with decoupling capacitors (C3 and C4) to stabilize the power supply and filter noise.

- The ground pin (GND) is connected to the circuit's common ground.

- **Connections:**
  - PB0–PB6 (Pins 12–18): Control the segments (A–G) of the two 7-segment displays.
  - PC0–PC1 (Pins 23, 24): Are used to multiplex the displays by activating the common cathode of each display (U3, U4) alternately.
  - PC2–PC3 (Pins 25, 26): Connected to the tactile switches for user input.

2. Tactile Switches
   - **Dice Type Selection**:
     - Connected to PC2 of the microcontroller.
   - **Dice Roll Switch**:
     - Connected to PC3 of the microcontroller.
   - **Pull-Down Resistors**:
     - Each switch is connected to ground through pull-down resistors (R13 and R14).

3. 7-Segment Displays (U3, U4)
   - **Display Type**:
     - Common cathode displays are used for showing two-digit dice roll results.
   - **Segment Control (Pins A–G)**:
     - Each segment is connected to a GPIO pin of the microcontroller (PB0–PB6) through current-limiting resistors (R4–R10, 250Ω each).

Task 8: Detailed Design and Planning – PCB Design

The project has reached the PCB design stage, with a finalized layout that integrates all components and adheres to the project's compact size constraints. The PCB measures 38x48 mm, fitting the key-fob size requirement, while ensuring functionality, durability, and manufacturability.

Final PCB (All Layers)

The image below represents the completed PCB design, showing all layers and components. It includes:

- Front copper traces connecting components.
- Back copper traces for signal routing and ground planes.
- Placement of all components such as the microcontroller, resistors, capacitors, LEDs, switches, and 7-segment displays.

This view provides a complete picture of how the circuit is routed, ensures optimal space utilization within the small dimensions, and remarks the dimensions of the PCB.

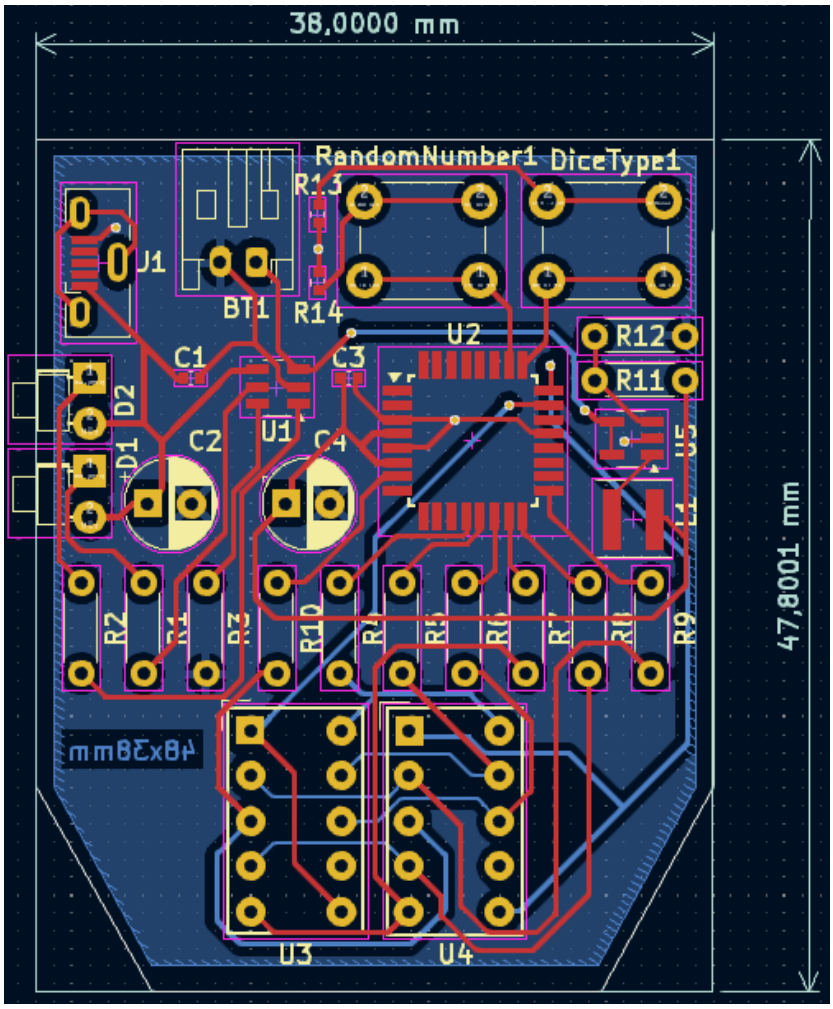


Figure 11. Electronic Dice PCB

Front Copper Layer

The second image highlights only the front copper layer, where most connections and signal routing occur. This layer focuses on:

- Power distribution from the battery and voltage regulator to components.
- Critical signal traces connecting the microcontroller to the 7-segment displays and switches.

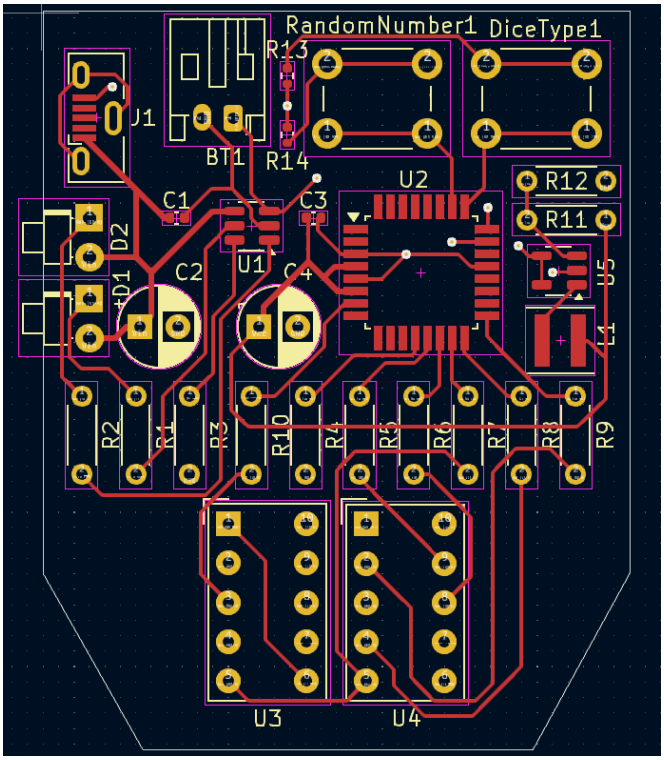


Figure 12. Front Copper Layer

Back Copper Layer

The third image isolates the back copper layer, primarily used for:

- Ground planes to reduce noise and stabilize the circuit.
- Supporting additional routing where space on the front layer is insufficient.

The inclusion of a back copper layer ensures a balanced and reliable PCB design, even with compact dimensions.

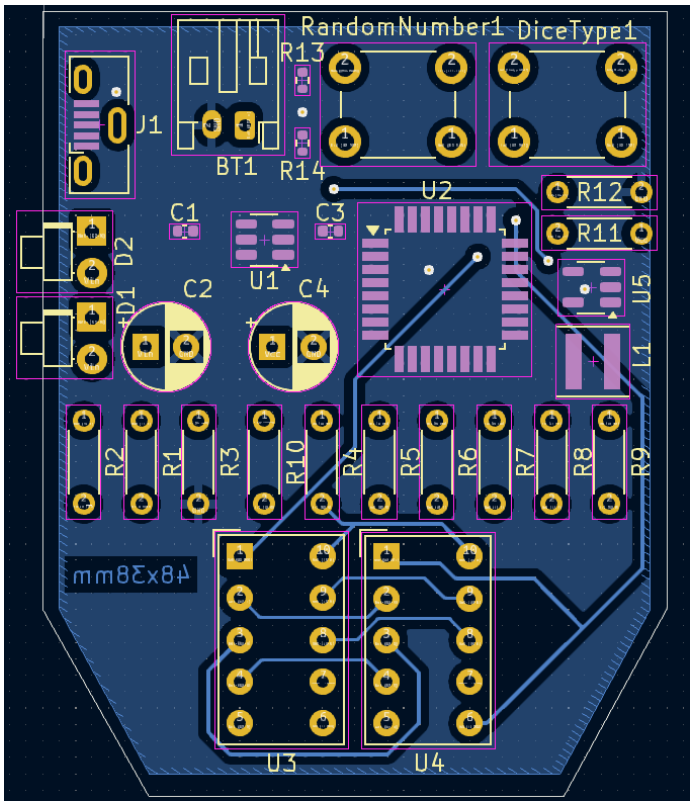


Figure 13. Back Copper Layer

3D View

The fourth image provides a 3D rendering of the assembled PCB, showcasing the layout and physical placement of components. Key features include:

- Two 7-segment displays prominently positioned for clear visibility.
- Tactile switches for user input, aligned at the top for ergonomic access.
- Microcontroller and other passive components efficiently placed to minimize trace lengths.

This visualization confirms the design's practicality and highlights the compact arrangement within the 38x48 mm footprint.

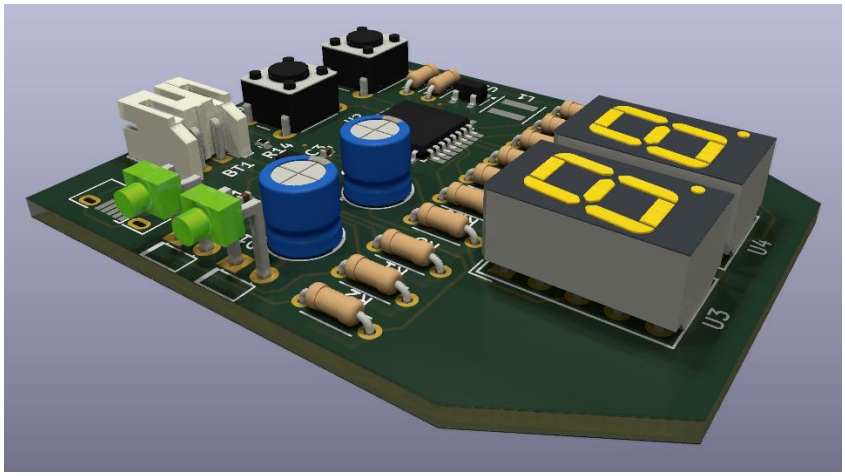


Figure 14. Electronic Dice PCB's 3D View

Task 9: Detailed Design and Planning – Material and Resource Planning

Using the approved seller's list from Task 2, I verified the availability of all components required for the electronic dice PCB design. Below is the finalized Bill of Materials (BoM), including each component's description, quantity, supplier, and total cost.

Product Name	Product Description	Distributor	Manufacturer	Cost (inc. VAT)	Quantity	Total
ATMEGA328P-AU	8-bit Microcontrollers - MCU 32KB In-system Flash 20MHz 1.8V-5.5V	Mouser Electronics	Microchip	£2.10	1	£2.10
7-segment (HDSP-513A)	LED Displays & Accessories Red 643nm 0.56in 7 Segment	Mouser Electronics	Broadcom Limited	£1.55	2	£3.10
250R Resistor (OK15G5E-R52)	Carbon Film Resistors - Through Hole 1.5 Ohm 1/4W 5% 250 Volt	Mouser Electronics	Ohmite	£0.09	7	£0.63
Tactile Switch (PHAP3344R)	Tactile Switches TACT SWITCH PCB MOUNT	Mouser Electronics	Apem	£0.32	2	£0.64
TP4057	SOT23-6 BATTERY MANAGEMENT ICS R	DigiKey	UMW	£0.42	1	£0.42
Usb B Micro (USB3080-30-00-A)	USB Connectors Micro B Skt, Bottom-SMT, R/A, 30u" NoPeg, W/shell stake, T&R	Mouser Electronics	GCT	£0.56	1	£0.56
JST-PH 2 Pin Connector (B2B-PH-K-S)	CONN HEADER VERT 2POS 2MM	DigiKey	JST Sales America Inc.	£0.16	1	£0.16
Led Charging (TLPR5600)	LED RED DIFF SIDE VIEW T/H R/A	DigiKey	Vishay Semiconductor Opto Division	£0.60	1	£0.60
Led Charged (TLPG5600)	LED GREEN DIFF SIDE VIEW T/H R/A	DigiKey	Vishay Semiconductor Opto Division	£0.60	1	£0.60
Ceramic Capacitor (CL05B103JB5NNND)	CAP CER 0.1UF 50V X7R 0402	DigiKey	Samsung Electro-Mechanics	£0.10	2	£0.20
Electrolytic Capacitor (865080642006)	CAP ALUM 10UF 20% 50V SMD	DigiKey	Würth Elektronik	£0.18	2	£0.36
1k Resistor (CF1/2CT52R102J)	RES 1K OHM 5% 1/2W AXIAL	DigiKey	KOA Speer Electronics, Inc.	£0.11	2	£0.22
2k Resistor (CF1/4CT52R202J)	RES 2K OHM 5% 1/4W AXIAL	DigiKey	KOA Speer Electronics, Inc.	£0.09	2	£0.18
Lipo Battery (M105318)	3.7V 150mAh LiPo battery.	Model Rockets	PKCell	£8.75	1	£8.75
Step-Up Voltage Regulator (LM2731YMF)	Switching Voltage Regulators 0.6/1.6 MHz Boost Converters With 22V Internal FET Switch in SOT-23 5-SOT-23 - 40 to 125	Mouser Electronics	Texas Instruments	£3.02	1	£3.02
1.2k Resistor (CFM12JT1K20)	RES 1.2K OHM 5% 1/2W AXIAL	DigiKey	Stackpole Electronics Inc	£0.08	1	£0.08

3.9k Resistor (CF14JT3K90)	RES 3.9K OHM 5% 1/4W AXIAL	DigiKey	Stackpole Electronics Inc	£0.08	1	£0.08
SMD Power inductor (SRR1210-100M)	Power Inductors - SMD 10uH 20% SMD 1210	Mouser Electronics	Bourns	£0.96	1	£0.96
10k Resistor (MCT0603MD1052BP500)	Thin Film Resistors - SMD MCT 0603-25 0.1% AT P5 10K5	Mouser Electronics	Vishay / Beyschlag	£0.34	2	£0.68
				Total		**£23.34**

Table 4. Materials Needed

Observations

1. Suppliers:
   - Most components are sourced from Mouser Electronics and DigiKey due to their reliability and comprehensive selection.
   - The LiPo battery was sourced from Model Rockets, a trusted supplier for compact power sources.

2. Availability:
   - All components are available and compatible with the PCB design. There is no risk of delay due to unavailability.

3. Cost Efficiency:
   - The total cost is reasonable for a functional prototype, staying within budget constraints.

This finalized BoM ensures all necessary components are ready for assembly and testing.

Task 10: Field Ready Prototype and Testing and Evaluation

In this task, I focused on testing the electronic dice system through a simulated breadboard setup using software tools. This practical simulation allowed for evaluating the functionality of key components, such as the buttons, random number generation, and the display of results on the 7-segment displays. By conducting the tests in a simulated environment, potential issues in the design were identified and resolved without the need for a physical prototype. This simulation served as a crucial step in validating the logic and performance of the circuit, ensuring that the system meets the intended design requirements.

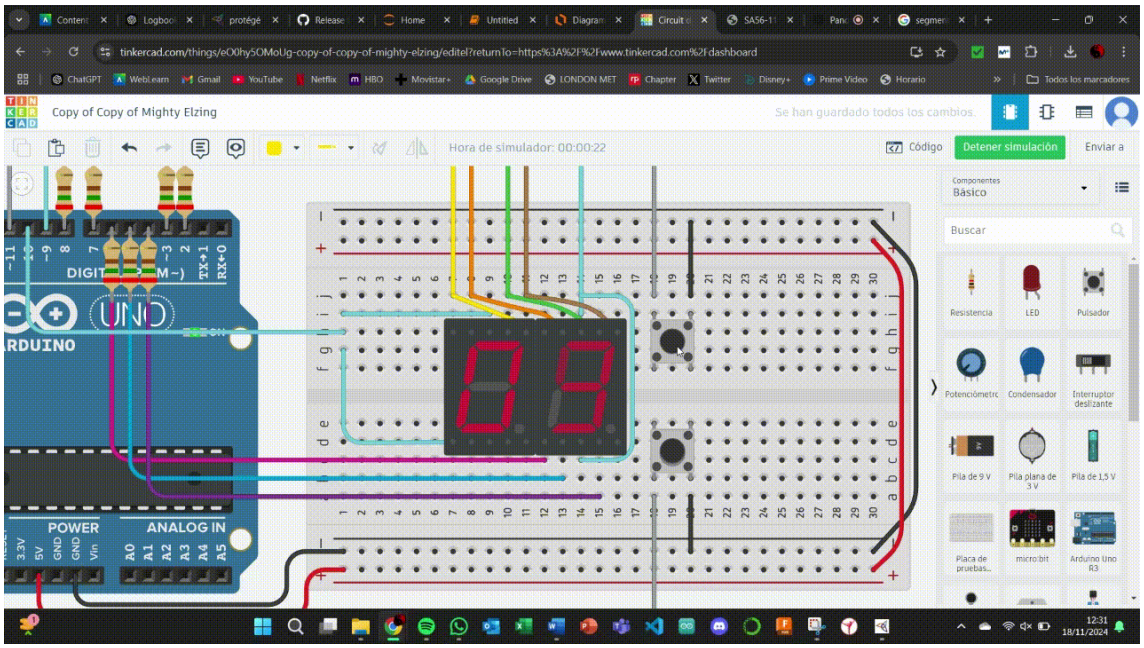


Figure 15. Field Ready Type Simulation

Task 11: Optimization and Improvement

The current design of the electronic dice meets the functional and size requirements, but there are several areas where it can be improved to enhance efficiency, size, and usability. Below are the proposed improvements:

Transition to SMD Components

One of the most effective ways to make the design smaller is by replacing through-hole components with Surface Mount Devices (SMD).

- Using SMD resistors and capacitors can significantly reduce the PCB size.
- Smaller components result in shorter traces, which improves signal integrity and reduces electromagnetic interference (EMI).

This change could potentially reduce the PCB dimensions below 38x48 mm, making the design more compact and portable.

Implement Sleep Mode

Currently, no sleep mode has been implemented, meaning the device will consume power continuously, even when idle. Adding a sleep mode could significantly improve battery life:

1. How to Implement:
    - Use the microcontroller's built-in sleep features to enter a low-power state when the device is inactive.
    - Wake the microcontroller only when a button press is detected.

2. Benefits:
    - Reduces power consumption during idle periods.
    - Extends the battery life, making the dice more energy-efficient and practical for long-term use.

Optimize Power Management

1. Current Design:
    - The step-up regulator (LM2731) ensures a stable 5V output but introduces inefficiency at low loads.

2. Proposed Improvement:
    - Consider switching to a more efficient regulator with a power saving mode or a buck-boost regulator to adapt to the varying load conditions of the dice.
    - Add a power switch to manually disconnect the battery when not in use.

Improve Display Brightness Control

The 7-segment displays are always powered at full brightness, which might be unnecessary in all lighting conditions.

- Implement PWM control for the displays to adjust brightness dynamically.
- Use a light-dependent resistor (LDR) to create an adaptive brightness feature based on ambient light.

Add sound feedback

A small piezo buzzer could provide an audible click or tone when the dice is rolled, improving the user experience.

Task 12: Design a 3D model case for the product

The case for the electronic dice has been designed with attention to detail to accommodate all components, ensure portability, and maintain durability. Below, the different perspectives of the design are analysed.

1. Initial view with only visible edges

This view highlights the basic shape and features of the case, including:

- Clean geometry with bevelled edges for a modern aesthetic.
- Two rectangular cutouts on the top for the dual 7-segment displays and the tactile switches.
- A dedicated compartment to store the battery, which includes a removable cover for easy access.
- Lateral holes for the USB connector, and the two LEDs.

- Legs so that the case can be standing on any plain surface.
- A compact design that prioritizes functionality while keeping the overall footprint minimal.

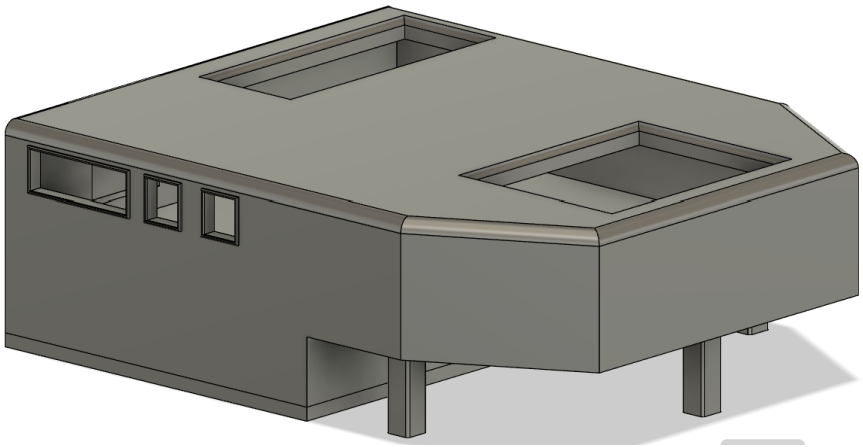


Figure 16. Electronic Dice's Case

2. Initial view with all edges

This transparent view allows the internal structure to be seen:

- Internal Component Layout: The positions of internal supports and mounting structures for the PCB and components are clearly visible.
- Precision Cutouts: Openings for the USB port, tactile switches, and other components are visible and aligned with their respective functionalities.

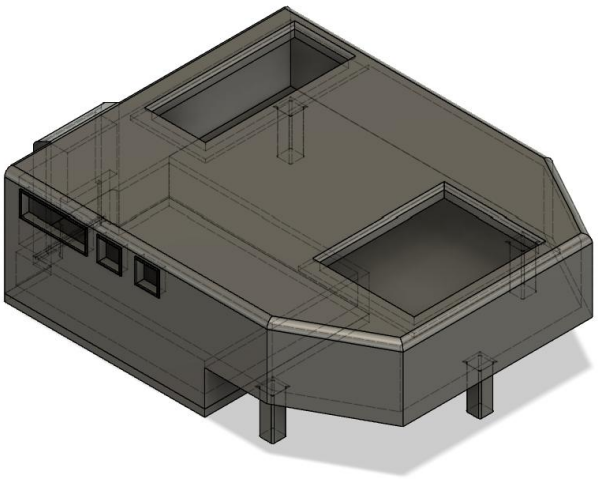


Figure 17. Case With All Edges Visible

3. Frontal view

The frontal view shows:

- The two cutouts for the 7-segment displays and the tactile switches, ensuring they are flush with the surface for clear visibility.
- A smooth, symmetrical design that maintains simplicity and ease of use.

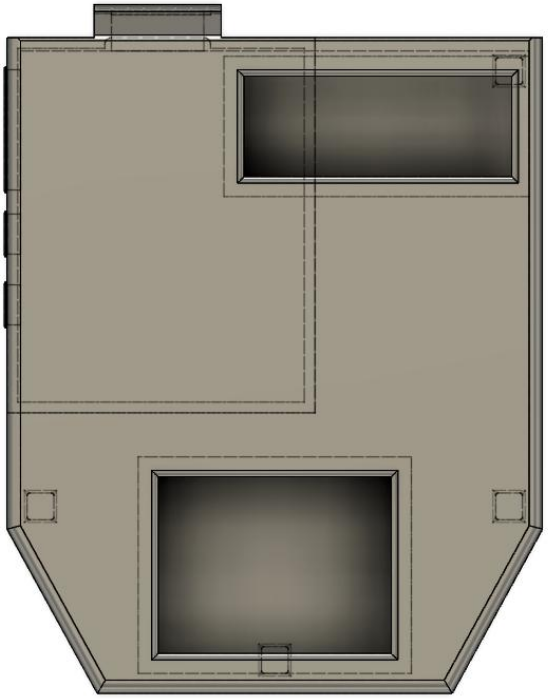


Figure 18. Frontal View

4. **Top rear corner view**

This perspective provides insight into:

- The angular design of the rear section, which accommodates the LiPo battery.
- Additional mounting posts and structural elements that reinforce the case for durability and equilibrium.

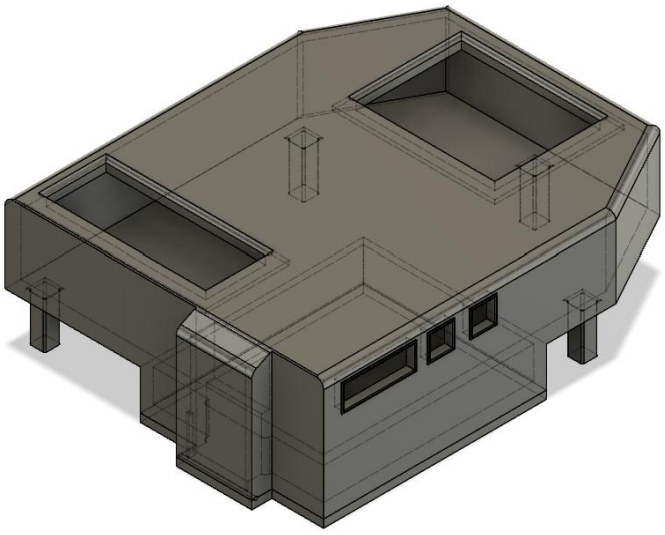


Figure 19. Top Rear Corner View

5. **Top schematic with dimensions**

This schematic provides precise measurements of the case:

- Overall Dimensions: The case measures 40.2 mm x 51.5 mm, ensuring it remains compact and portable.

- Cutout Sizes: Detailed dimensions for the 7-segment display cutouts and side openings show careful planning for component placement.

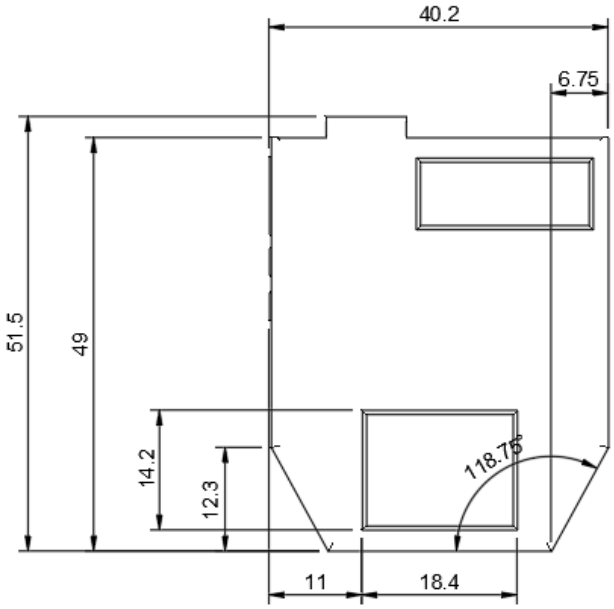


Figure 20. Top Schematic

6. Frontal and right-side schematic with dimensions

This view offers a detailed look at:

- The height of the case (16 mm), ensuring sufficient clearance for internal components.

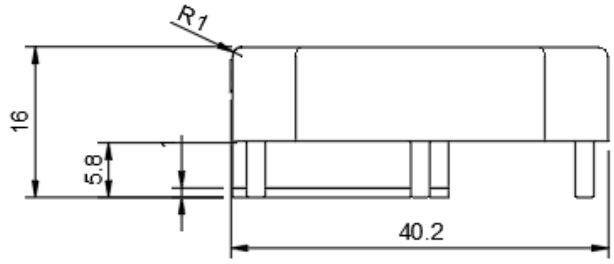


Figure 21. Frontal Scheme

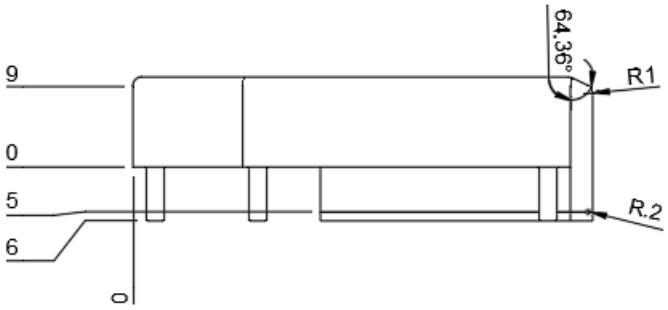
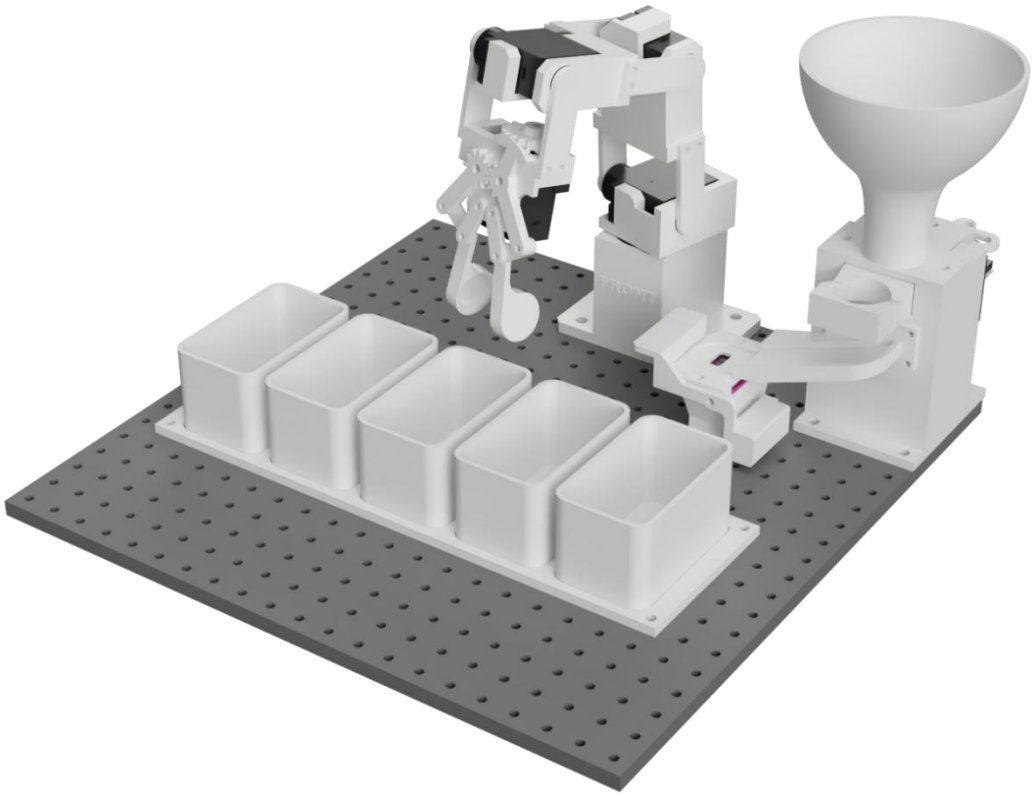


Figure 22. Right-Side Schematic

5DoF Robotic Arm – Group Project



Authors:
Pawel Toborek (19024794)
Jorge Viñuela (23038621)
Saksham Arora (23022498)
Olayinka Afolabi (23054001)

Supervisors:
Marcio Lacerda
Dion Mariyanayagam
Lynn Crothall

Table of Contents

Table of Figures

List of Tables

Abstract

This project outlines the design and execution of a 5-degree-of-freedom (5-DoF) robotic arm, including mechanical, electrical, and software components to provide accurate, automated item manipulation. The robotic arm, driven by an ESP32 microcontroller, has servo motors, a color sensor, and an accessible web-based interface for seamless control and operation. The primary functions encompass forward and inverse kinematics, allowing precise movement and placement, alongside a gripper mechanism refined for manipulating tiny items like Skittles.

The arm's capacity to categorize things by color was achieved using sophisticated kinematic algorithms, accurate motion control, and instantaneous data processing. Safety methods were instituted to improve dependability, encompassing joint angle restrictions, motion smoothing, and effective error management. The idea utilized economic materials and 3D-printed elements, guaranteeing accessibility and affordability.

The results indicate the arm's efficacy as a multifunctional platform for educational applications, research, and minor industrial operations. This study highlights the promise of affordable, high-performance robotic systems, establishing a basis for future progress in robotics and automation.

I. Introduction

Objective

The objective of this project is to design and develop a 5-degree-of-freedom robotic arm using an ESP32 microcontroller. This project aims to:

1. Build and optimize the robotic arm's structure, including a high-precision gripper.
2. Implement kinematic algorithms to enable accurate control and movement.
3. Develop a user-friendly, web-based interface for intuitive control of the robotic arm.
4. Integrate wireless connectivity to allow remote access and operation.

By achieving these objectives, the project seeks to create a versatile robotic arm suitable for tasks requiring precision and adaptability, while providing a learning platform for robotics and control systems.

Problem Statement

In various fields such as manufacturing, research, and education, there is a growing need for adaptable and precise robotic systems that can perform complex tasks with minimal human intervention. Traditional robotic arms often lack affordable control solutions with advanced features like wireless connectivity, customizable kinematics, and user-friendly interfaces, which limits their accessibility and usability in smaller-scale or educational contexts [1,2].

This project addresses these issues by developing a low-cost, 5-degree-of-freedom robotic arm that uses an ESP32 microcontroller. The arm will be controlled via an intuitive web-based interface, allowing easy, remote operation and control. Additionally, by optimizing the gripper and implementing kinematics, this project aims to enhance the robotic arm’s precision, making it suitable for applications requiring accurate manipulation and positioning. The project therefore aims to bridge the gap between functionality, affordability, and accessibility in robotic arm systems.

Relevance of the project

This project holds significant relevance as it combines multiple disciplines—mechanical design, electronics, programming, and kinematics—to create a versatile robotic arm [7]. By using a 5-degree-of-freedom structure, the arm can achieve a high level of manoeuvrability, which is essential for real-world applications like precise object manipulation and sorting. Leveraging the ESP32 microcontroller offers a cost-effective, powerful platform with built-in Wi-Fi, enabling remote control through a user-friendly web interface. This connectivity expands potential uses, from industrial automation to remote education and research applications [8,9,10].

Furthermore, optimizing the gripper design and implementing kinematic algorithms are crucial for achieving accurate and responsive movement. These capabilities enhance the arm’s ability to perform tasks with precision, which is valuable for robotics research and provides students with

hands-on experience in solving complex, real-world engineering challenges. This project thus not only serves as an educational tool but also lays the foundation for advancements in robotic control systems and applications.

Background

Robotic Arm

A robotic arm is a programmable mechanical device that mimics the motion and functionality of a human arm. It typically consists of multiple segments, or links, connected by joints that allow for various degrees of freedom (DoF). Robotic arms are commonly used in industries such as manufacturing, medical technology, and research, where precise, repeatable movements are required. In this project, we aim to design a 5-degree-of-freedom robotic arm capable of performing precise tasks such as sorting and object manipulation. Developing this robotic arm requires careful planning of its kinematic structure, control systems, and end-effector (or gripper) design to ensure both accuracy and reliability [1,3,4,5].

Servo MG90S

The MG90S is a compact, high-torque servo motor commonly used in robotics applications due to its affordability, lightweight design, and performance [13]. It features a metal gear train that ensures durability and precision, making it ideal for applications where high torque and precision control are necessary. Each joint in the robotic arm is driven by an MG90S servo motor, which allows for fine-grained control of the arm's movement across its various degrees of freedom. These servo motors are critical for achieving the precision required for tasks such as sorting and manipulating objects [6,7,8].

ESP32

The ESP32 is a powerful microcontroller with integrated Wi-Fi and Bluetooth capabilities, making it a popular choice for IoT and robotics applications [12]. Its high processing power, real-time control capabilities, and connectivity options make it ideal for controlling robotic systems like the 5-DoF robotic arm in this project. The ESP32 allows for precise motor control, easy integration of sensors, and wireless communication. This wireless capability is particularly valuable for developing a web-based interface, allowing users to control the robotic arm remotely, enhancing usability and accessibility [9,4,3].

Inverse Kinematics

Inverse kinematics (IK) is a mathematical approach used to calculate the required joint angles of a robotic arm to position its end effector at a specific location and orientation in space. In contrast to forward kinematics, which determines the position of the end effector based on known joint angles, inverse kinematics works backward, starting from the desired endpoint to determine the necessary joint parameters. IK is crucial for applications requiring precision, as it enables the robotic arm to perform complex, accurate movements, allowing it to reach specific points in its workspace. Implementing inverse kinematics in this project will provide the robotic arm with the ability to accurately move and position its gripper, enhancing its usefulness for tasks that require exact positioning, such as sorting or picking objects [1,6,8,5].

II. Hardware design

The hardware design of the robotic arm incorporates a variety of mechanical and electrical components, ensuring the functionality, stability, and precision required for its operation. This section details the components used in the design, including a Bill of Materials (BOM) and a breakdown of the 3D-printed parts.

Bill of Materials (BOM)

The table below outlines the components required to assemble the arm, which includes electronic components, servo motors, and various fasteners and bearings. The BOM ensures that all necessary components are accounted for and provides flexibility in sourcing, as it includes detailed specifications for each item.

Nº	P/N	Link	Price	Items
1	Torx Screw Set	https://amzn.eu/d/iVCb0gH	£7.99	Set 760 Pieces
2	6x10x3mm Bearings	https://amzn.eu/d/bLbZnPb	£7.99	10
3	7x22x8mm Bearing	https://amzn.eu/d/cW2G3Cp	£7.09	8
4	TCS34725	https://amzn.eu/d/2pwwxkY	£4.99	1
5	Servo MG90S	Provided	-	6
6	ESP32 Devkit V1	Provided	-	1
7	PCA9685	Provided	-	1
8	SD Card Reader	Provided	-	1

Table 5. Bill Of Materials

3D-Printed parts:

The arm relies heavily on custom-designed 3D-printed parts for its structural integrity. These parts are essential for creating the joints, segments, and gripper mechanisms that facilitate movement and manipulation tasks. Important to remark that the whole robot was redesigned, not only the gripper. The following table provides a detailed breakdown of the required 3D-printed parts.

Nº	Name	Count
1	Base	1
2	Segment1 Base	1
3	Motor Shaft Swing	3
4	Bearing Swing	3
5	Segment2 Base	2
6	Gripper Base	1
7	Gripper Main Gear	1

8	Gripper Left Gear Arm	1
9	Gripper Right Gear Arm	1
10	Gripper Left Link	1
11	Gripper Right Link	1
12	Gripper Left Clamp	1
13	Gripper Right Clamp	1

Table 6. 3D-Printed Parts

Figure 1 shows the 3D-printed components required for the assembly of the robotic arm, each part is positioned for optimal printing, and the preview highlights details such as supports, infill, and walls. The visible parts include bases, joint connectors, motor holders, and the gripper mechanism, all essential for the arm's structural integrity and functionality. The slicing summary indicates a total filament usage of approximately 58.45 grams and an estimated print time of 2 hours.

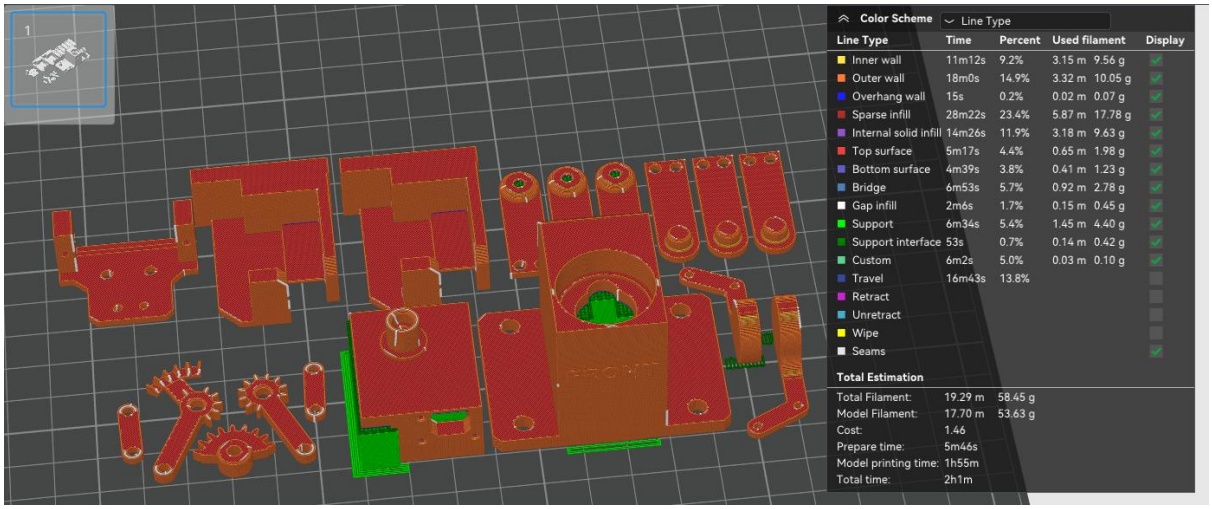
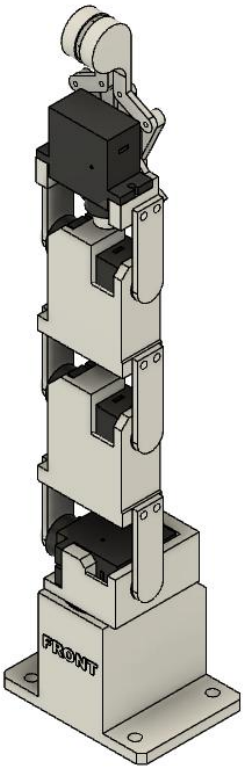


Figure 23. 3D-Printed Parts Layout and Printing Details

Figure 24. 3D Printed Parts

Assembly and Integration

Finally, the integration joins electronic components and 3D-printed parts into a cohesive system. The servos (MG90S) are controlled via the ESP32 microcontroller, which communicates with the PCA9685 servo driver [15]. The structural components are assembled using the screws, spacers, and bearings listed in the BOM, ensuring stability and minimizing wear during operation. The following figure illustrates the complete assembly of all components that form the arm and provides two perspectives of the arm as modelled in Autodesk Fusion 360 [11], showcasing the structural design and arrangement of parts in a 3D environment.



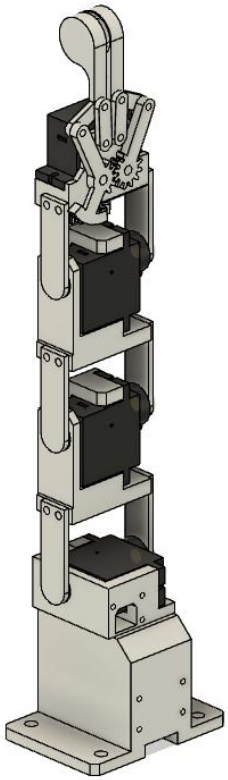


Figure 25. Robotic Arm Assembly (Front)

Figure 26. Robotic Arm Assembly (Back)

Wiring Diagram

This section illustrates the electrical connections and integration of the core components, by providing a detailed overview of how the ESP32 microcontroller interfaces with the peripheral modules, including the PCA9685 PWM driver, the TCS34725 colour sensor [14], and the SD card module. Proper wiring ensures efficient communication between components, power delivery, and synchronization, which are critical for the robotic arm's functionality and performance.

Firstly, the ESP32 Devkit V1 is the central microcontroller in the system and is powered by both 5V and 3.3V supplies. The connections include:

- **SDA and SCL pins (I2C communication):** connected to peripherals such as the PCA9685 PWM driver and the TCS34725 colour sensor.

- **MOSI, MISO, and SCK pins (SPI communication)**: primarily interfacing with the SD card reader module.
- **SD_EN pin**: used to enable or control the SD card module.

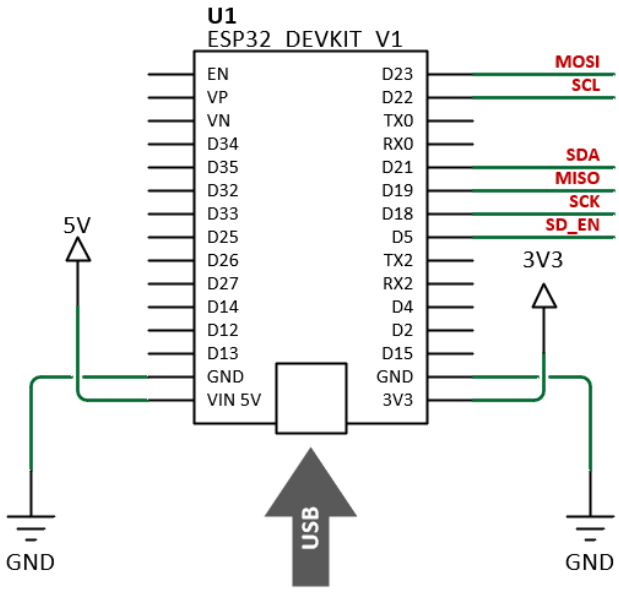


Figure 27. ESP32 wiring diagram

The PCA9685 servo driver is connected to the ESP32 via the I2C, operates at 3.3V logic levels and drives up to 16 channels of servos.

- **Channels CH0 to CH5:** used for controlling six servo motors (SERVO_0 to SERVO_5).
- **V+ pin:** supplies power to the servos, which is independent of the PCA9685's operating voltage and is provided through an external 5V supply.

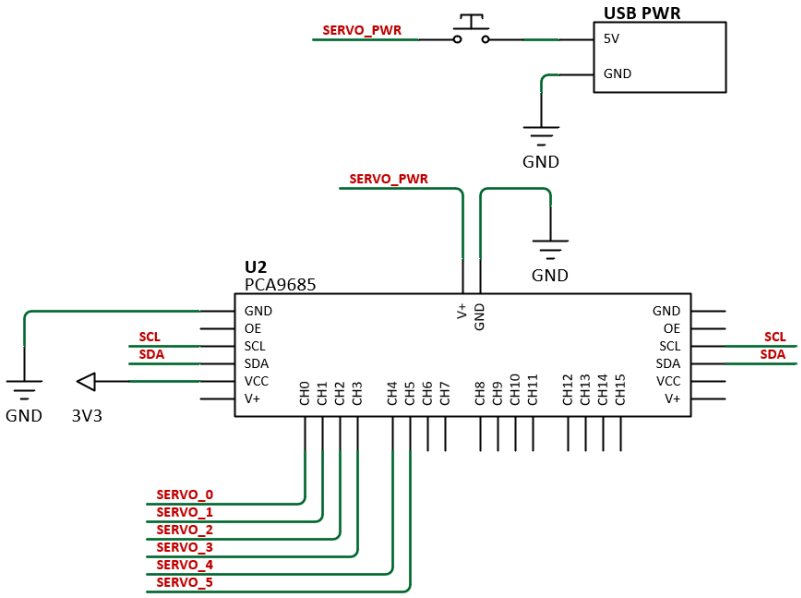


Figure 28. Servo driver wiring diagram

The TCS34725 colour sensor is also connected to the ESP32 using the I2C interface and operates at 3.3V, with:

- **SDA and SCL** lines linked to the ESP32 for data communication.
- **VIN and GND** connected to power and ground, ensuring proper operation of the sensor.

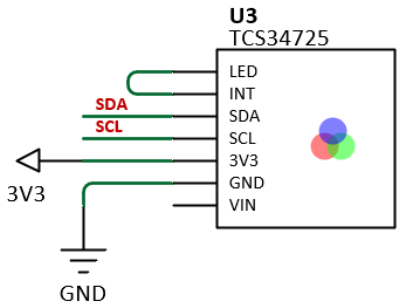


Figure 29. Colour sensor wiring diagram

Finally, the SD Card module interfaces with the ESP32 using SPI communication:

- **MOSI, MISO, SCK, and SD_EN pins:** connected to their respective pins on the ESP32.
- **VIN and GND** connected to power and ground, ensuring proper operation of the sensor, same as in the previous component.

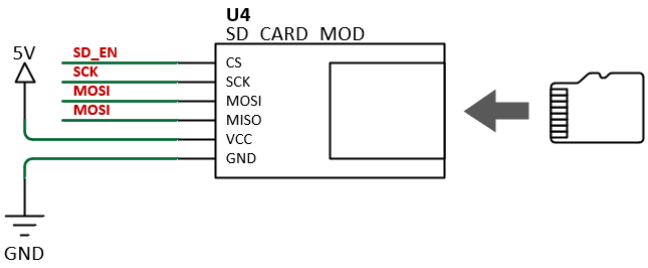


Figure 30. SD Card wiring diagram

III. Software Design

Robotic Arm

This section describes the arm's different functionalities, showcasing its ability to perform precise and complex movements through advanced algorithms and robust control mechanisms. It is equipped with servo motor control for seamless and accurate joint movements, inverse kinematics for calculating joint angles to achieve specific end-effector positions, and G-code support for programmable, task-specific actions. Additionally, the integration of a colour sensor enables dynamic object detection and sorting, making the arm versatile for a range of applications.

Each functionality is designed to operate cohesively, providing a reliable and efficient system that can adapt to real-world tasks requiring precision and automation. Below, detailed explanations and code snippets highlight the implementation of these functionalities, illustrating how they contribute to the arm's overall performance.

Arm Definition

The robotic arm is implicitly defined through the combination of servo motor channels managed by the Adafruit_PWMServoDriver and the control logic that governs their movements. The following elements together define the arm:

1. **Servo Channels:** The arm's joints are represented by up to 16 servo channels on the PCA9685 PWM driver, as instantiated here:

```
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x40);
Adafruit_TCS34725 tcs = Adafruit_TCS34725(TCS34725_INTEGRATIONTIME_2_4MS, TCS34725_GAIN_4X);
```

Figure 31. Servo Channels Definition

2. **Arm's links:** The arm's structure and movement are governed by inverse kinematics algorithms, as seen in the solveIK3D and solveIK functions. These functions define the geometry of the arm using linkLengths and calculate the required joint angles:

```
float linkLengths[] = {50.0, 50.0, 75.0};
float linkAnglesIK[] = {-1.57, 0, 1.57};
const int numLinks = 3;
const float tolerance = 0.5;
```

Figure 32. Links Definition

3. **Servo Positions:** The current and goal positions of the servos are stored in arrays:

```
int servoPos[16];
int servoPosGoal[16];
```

Figure 33. Servo Positions Definition

Servo and Speed Control

The robotic arm uses PWM signals to control the angles of the 6 servo motors. The servo positions are mapped from angles in degrees (0–180) to PWM pulse lengths, where each servo is individually addressed, allowing precise control of joint angles.

```
void setServoPos(uint8_t n, float angle) {
  if (n < 0 || n >= 16 || angle < 0 || angle > 180) {
    Serial.println("Invalid servo ID or angle");
    return;
  }

  uint16_t pwmVal = map(angle, 0, 180, SERVOMIN, SERVOMAX);
  pwm.setPWM(n, 0, pwmVal); // Send PWM signal to servo
  servoPosGoal[n] = pwmVal;

  Serial.print("Servo ");
  Serial.print(n);
  Serial.print(" set to: ");
  Serial.print(angle);
  Serial.print(" deg (PWM: ");
  Serial.print(pwmVal);
  Serial.println(")");
}
```

Figure 34. SetServoPos function

The “setServoPos” function controls the position of a servo motor by converting a given angle into the corresponding PWM signal, which is then sent to the servo using the Adafruit PCA9685 driver. It validates the input to ensure the servo ID and angle are within acceptable ranges and then uses the map function to translate the angle into a PWM pulse length, ensuring the servo moves to the desired position. The resulting PWM value is stored in the “servoPosGoal” array for tracking, and detailed logs are printed for debugging.

```
bool reachedGoad = true;
for(int i=0; i<16; i++){
  if(abs(servoPos[i] - servoPosGoal[i]) < 3){
    servoPos[i] = servoPosGoal[i];
  }else{
    if(servoPos[i] < servoPosGoal[i]){
      servoPos[i]+=2;
    }

    if(servoPos[i] > servoPosGoal[i]){
      servoPos[i]-=2;
    }
  }

  //check if all motors reached the goal
  if(servoPos[i] != servoPosGoal[i]){
    reachedGoad = false;
  }

  pwm.setPWM(i, 0, servoPos[i]);
}
busy = !reachedGoad;
digitalWrite(LED_PIN, busy);
```

Figure 35. Smooth servo movement implementation

Once the desired servo position has been obtained, the loop function ensures a smooth servo movement by gradually adjusting each servo's position toward its target, shown in figure 11. If the difference is small, the servo is directly set to the target; otherwise, its position is incrementally increased or decreased to avoid abrupt movements. The program tracks whether all servos have reached their targets using the “reachedGoad” flag, updating the busy state, accordingly, reflected on LED_PIN, which serves as a visual indicator of whether the servos are still in motion or idle.

Web interface

The website interface was written using HTML and JavaScript. To get the 3D visualization p5js library was used, as it provides easy graphical interface capable of 3D rendering using WEBGL. The website files are served by the ESP32 server inside the SD card with all necessary files. The 3D models are “.stl” files loaded by the JavaScript code.

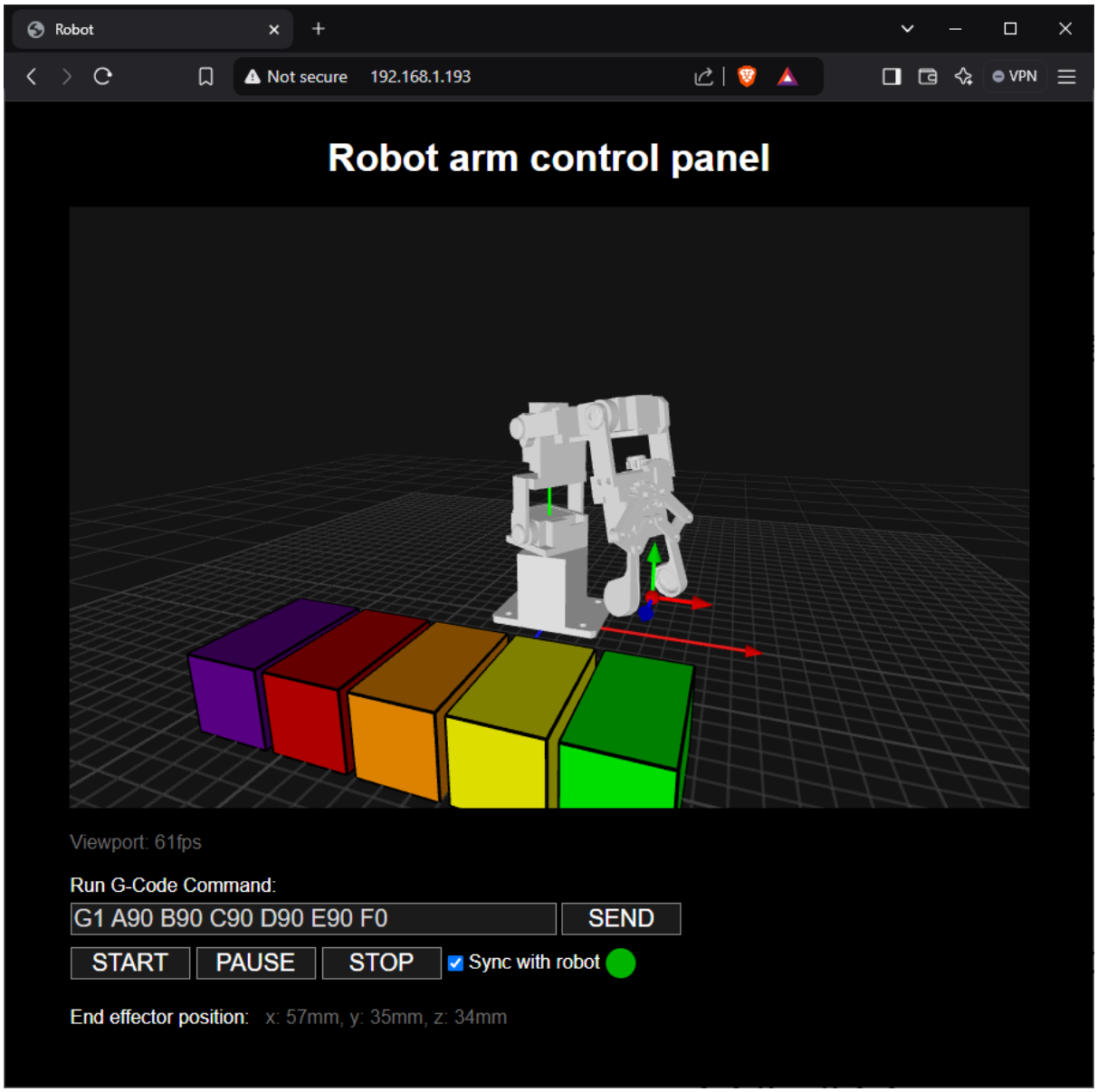


Figure 36. Robot arm control panel

The web interface includes a text field that allows the user to input G-Code commands to control the robotic arm. Several G-Code commands are supported, including G0, which accepts XYZ coordinates and utilizes inverse kinematics to move the arm to the specified position. Additionally, G1 enables direct control of the motor angles for each joint, offering precise adjustments for the arm's segments. This command also facilitates the operation of other servo channels, making it possible to integrate and manage additional hardware seamlessly.

Colour Sensor

The integration of the colour sensor adds a vital functionality to the robotic arm, enabling it to detect and classify objects (Skittles) based on their colour. The sensor (TCS34725) reads the red, green, and blue (RGB) components of an object and matches them to predefined categories stored in the “colorList” array. Each entry in the array represents the average RGB values for a specific colour, such as red, green, yellow, or purple.

```
ColorEntry colorList[] = {
  ColorEntry{93.25, 92.5, 62.5}, // Empty (no object detected)
  ColorEntry{111.5, 86.5, 62},   // Purple
  ColorEntry{150, 69.5, 50.5},   // Red
  ColorEntry{144, 70, 42},       // Orange
  ColorEntry{122, 90, 40.5},     // Yellow
  ColorEntry{93.5, 111.25, 51}   // Green
};
```

Figure 37. Colour List Definition

Each colour is represented by its red, green, and blue intensities, allowing the program to match sensor readings to these predefined values. Next, the program calculates the Euclidean distance between the sensor's current RGB readings (r, g, b) and each predefined colour in the list. The colour with the smallest distance is selected as the detected colour, provided the distance is within a defined threshold. This process is described in the following figure.

```
int findClosestColorId(float r, float g, float b) {
  float minDistance = 999999.0;  // Initialize to a large number
  int closestId = 0;  // Default to no match

  for (int i = 0; i < numColors; i++) {
    // Calculate the Euclidean distance between the input color and each color in the list
    float distance = sqrt(pow(r - colorList[i].r, 2) + pow(g - colorList[i].g, 2) + pow(b - colorList[i].b, 2));

    // Check if the distance is the smallest so far and within a reasonable threshold
    if (distance < minDistance && distance < 15) {
      minDistance = distance;
      closestId = i;  // Store the index of the closest color
    }
  }

  return closestId;
}
```

Figure 38. FindClosestColorId Function

These functions allow the robotic arm to continuously detect colours in real time, ensuring precise and accurate classification for the objects used in this project.

IV. Inverse kinematics and Safety Mechanism

Introduction to Inverse Kinematics

Inverse Kinematics (IK) is a fundamental concept in robotic control, enabling the calculation of joint angles to position an end-effector at a specified Cartesian coordinate. IK plays a critical role in improving the functionality of robotic arms by allowing precise, repeatable motions, essential for tasks like assembly, manipulation, and navigation in constrained environments. This section discusses the mathematical underpinnings of IK, the integration of safety mechanisms, and challenges and solutions in implementing these techniques [1][4].

Mathematical interpretation

The implementation used for the robot described in this report first converts 3D position to 2D position and then solves the IK in 2D space.

The 3D to 2D conversion is done as follows:

$$baseRot = atan2(T_z, T_x)$$
$$dist = \sqrt{{T_x}^2 + {T_z}^2}$$

$dist$ is the X coordinate in 2D space
Y coordinate is the same as in 3D space (T_y)
$baseRot$ is used as the angle for base servo motor.

Forward kinematics:
is the process of calculating the position and orientation of the end-effector based on the joint parameters (such as angles or displacements) of the links in a kinematic chain.

In 2D space this can be done using trigonometric equations:

$$x = \cos(\theta_1) * len_1 + \cos(\theta_1 + \theta_2) * len_2 + \cos(\theta_1 + \theta_2 + \theta_3) * len_3$$
$$y = \sin(\theta_1) * len_1 + \sin(\theta_1 + \theta_2) * len_2 + \sin(\theta_1 + \theta_2 + \theta_3) * len_3$$

If necessary, the x and y coordinates can be translated by the robot origin position. The forward kinematics algorithm will use forrard kinematics to solve IK with CCD.

2D Inverse Kinematics (CCD):

The Cyclic Coordinate Descent (CCD) algorithm iteratively adjusts each joint of the kinematic chain, starting from the end effector (the last joint) and working backward toward the root joint.

At each iteration, the algorithm modifies one joint's angle to minimize the distance between the end effector and the target.

First, the algorithm checks the distance between the target and current position of the robot's end-effector. If it is less than the threshold the algorithm returns current motor positions. If the target is not reached, the algorithm calculates the angle to the joint by determining the vector between the joint and the end-effector, as well as the vector between the joint and the target. It then computes the angular difference between these vectors and adjusts the joint's angle accordingly to reduce the distance to the target. This process is repeated iteratively for each joint in the chain until either the target is reached within the specified threshold, or the algorithm reaches its iteration limit.

Code Implementation

Forward Kinematics

The "calculatePosition" function computes the Cartesian position of a specified point in the robotic arm, such as an intermediate joint or the end-effector. It uses the angles of the joints and the lengths of the links in the arm to iteratively calculate the cumulative position of the arm in 2D space. This function is essential for determining the spatial coordinates of any point along the arm, which is crucial for both visualizing the arm's configuration and performing inverse kinematics calculations.

```
Position calculatePosition(int iter) {
  float x = 0;
  float y = 0;
  float angleSum = 0;

  for (int i = 0; i < iter && i < numLinks; i++) {
    angleSum += linkAnglesIK[i];
    x += linkLengths[i] * cos(angleSum);
    y += linkLengths[i] * sin(angleSum);
  }

  Position result = {x, y};
  return result;
}
```

Figure 39. CalculatePosition function

It iterates through the arm's links up to the specified joint (iter), summing joint angles to determine the arm's cumulative orientation. For each link, it calculates the horizontal and vertical contributions using trigonometric functions scaled by the link lengths, adding these to the running totals. Once all links are processed, the function returns a Position struct with the calculated x and y coordinates, enabling the determination of any joint or end-effector position for motion planning and control. This function is essential for inverse kinematics as it calculates the current position of the end-effector or joints, providing the necessary feedback to iteratively adjust angles and minimize the distance to a target.

Inverse Kinematics

This section explores the implementation of inverse kinematics (IK) for controlling the robotic arm, enabling it to precisely position its end-effector at any target within its workspace. The IK process involves breaking down complex 3D movements into individual joint angles using geometric calculations and algorithms like Cyclic Coordinate Descent (CCD). To achieve this, there are two main functions that we have implemented: solveIK and solveIK3D.

1. SolveIK

This function computes the inverse kinematics for a 2D planar arm using the Cyclic Coordinate Descent (CCD) algorithm. Starting from the end-effector, it iteratively adjusts the angles of each joint to minimize the distance between the end-effector and the target position. Then, the CCD algorithm calculates the angle between two vectors: one from the joint to the end-effector and another from the joint to the target and uses this angle to modify the joint's position and bring the end-effector closer to the target.

```
Position jointPos = calculatePosition(i); // Position of the current joint

// Vectors
float jointToEffectorX = currentPos.x - jointPos.x;
float jointToEffectorY = currentPos.y - jointPos.y;
float jointToTargetX = targetX - jointPos.x;
float jointToTargetY = targetY - jointPos.y;

// Normalize vectors
float effectorLength = sqrt(jointToEffectorX * jointToEffectorX + jointToEffectorY * jointToEffectorY);
float targetLength = sqrt(jointToTargetX * jointToTargetX + jointToTargetY * jointToTargetY);

jointToEffectorX /= effectorLength;
jointToEffectorY /= effectorLength;
jointToTargetX /= targetLength;
jointToTargetY /= targetLength;

// Compute angle between vectors
float dot = jointToEffectorX * jointToTargetX + jointToEffectorY * jointToTargetY;
float det = jointToEffectorX * jointToTargetY - jointToEffectorY * jointToTargetX;
float angleChange = atan2(det, dot);

// Update joint angle
linkAnglesIK[i] += angleChange;
```

Figure 40. SolveIK3D function

The process is repeated for each joint in the chain, working backward from the end-effector to the base and continues iterating until the target is reached within a specified tolerance or the iteration limit is exceeded. This function ensures that the arm's joints are positioned correctly to achieve the desired target while respecting physical constraints, such as joint angle limits.

2. SolveIK3D

This next function handles the high-level logic for solving inverse kinematics in a 3D space. Firstly, it adjusts the y coordinate to account for the arm's geometry and calculates the base rotation angle to align the arm's base with the target position in the XZ plane, which reduces the target to a planar distance in this plane, effectively converting the 3D problem into a 2D problem. Secondly, the solveIK function previously explained, is called to compute the joint angles in the 2D plane, which are crucial for positioning the end-effector.

```
// Print the joint angles
for (int i = 0; i < numLinks; i++) {
  Serial.print("Angle ");
  Serial.print(i + 1);
  Serial.print(": ");
  Serial.print(linkAnglesIK[i]);
  Serial.print("rad, ");
  if(i==0){
    Serial.print(mymap(linkAnglesIK[i], -M_PI, 0, 0, 180)); //conver to deg
  }else{
    Serial.print(mymap(linkAnglesIK[i], -M_PI/2, M_PI/2, 0, 180)); //conver to deg
  }
  Serial.println("deg");
}

setServoPos(0, br);
setServoPos(1, mymap(linkAnglesIK[0], -M_PI, 0, 180, 0));
setServoPos(2, mymap(linkAnglesIK[1], -M_PI/2, M_PI/2, 180, 0));
setServoPos(3, mymap(linkAnglesIK[2], -M_PI/2, M_PI/2, 180, 0));
```

Figure 41. SolveIK function

After solving the 2D IK, the function maps the calculated base rotation and joint angles to servo-compatible values using the "mymap" function, which scales the input angles from their original range to the range required by the servos. Finally, it sets the positions of the base and joint servos using setServoPos. This function serves as the primary interface for positioning the arm in 3D space.

G-Code Integration

The control system incorporates a G-code interface, enabling the user to program and execute precise sequences of movements and actions through G-code commands, which provide a standardized way to communicate instructions to the arm, offering flexibility and ease of use for complex tasks. To achieve this, the "runGCodeCommand" function was implemented, which parses each command to determine its type (G0 or G1) and extracts the parameters needed for execution. Two G-code commands were added to the project: G0 for inverse kinematics, and G1 for forward kinematics.

1. **G0 Command**

   It is used to move the arm to a specific position in the space by specifying XYZ coordinates. Upon receiving a G0 command, the system invokes the solveIK3D function, which calculates the required joint angles and allows the arm to seamlessly reach the target position while respecting its physical constraints. Figure 15 shows the implementation of the "handleG0Command" function.

```
void handleG0Command(const String& command) {
    // Map motor labels to motor numbers
    const uint8_t motorMap[] = {'X', 'Y', 'Z', 'E'};
    float x = 0;
    float y = 0;
    float z = 0;
    float e = 0;
    // Loop through each motor label
    for (uint8_t i = 0; i < 4; i++) {
        bool found = false;
        double val = extractValue(command, motorMap[i], found);
        if (found) {
            if(motorMap[i] == 'X') x = val;
            if(motorMap[i] == 'Y') y = val;
            if(motorMap[i] == 'Z') z = val;
            if(motorMap[i] == 'E') e = val;
        }
    }
    Serial.println("----");
    Serial.print("Solving IK for position ");
    Serial.print("X:");
    Serial.print(x);
    Serial.print(", Y:");
    Serial.print(y);
    Serial.print(", Z:");
    Serial.println(z);

    solveIK3D(x, y, z);

    Position p = calculatePosition(numLinks);
    Serial.print("Reached to X:");
    Serial.print(p.x);
    Serial.print(", Y:");
    Serial.println(p.y);
}
```

Figure 42. HandleG0Command function

- **Extracting Target Coordinates:** It parses the G-code command to extract `X`, `Y`, `Z`, and `E` values, representing the target 3D position and an optional parameter.
- **Solving Inverse Kinematics:** It passes the extracted `x`, `y`, and `z` coordinates to the `solveIK3D` function, which computes the joint angles necessary to move the arm's end-effector to the specified target position.
- **Debugging and Verification:** The function prints the extracted coordinates and the arm's final calculated position to ensure the movement is executed as expected.

2. G1 Command

It enables direct control of the arm by specifying individual motor angles for each joint (forward kinematics). This command was particularly useful for fine-tuning the movements and creating the loop of sorting the skittles.

```
void handleG1Command(const String& command) {
    // Map motor labels to motor numbers
    const uint8_t motorMap[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P'};

    // Loop through each motor label
    for (uint8_t i = 0; i < 16; i++) {
        bool found = false;
        double angle = extractValue(command, motorMap[i], found);
        if (found) {
            // Set the servo position for the specific motor
            setServoPos(i, angle);
        }
    }
}
```

Figure 43. HandleG1Command function

- **Extracting Motor Angles:** The function iterates through the motorMap and retrieves the specified angle from the command string. If a valid angle is found, the found flag is set to true, and the angle is stored.
- **Setting Servo Positions:** If a valid angle is extracted for a motor, the setServoPos function is called, which updates the corresponding servo to the specified position, allowing precise control of each motor based on the command input.

Viable Safety Features

There are a series of safety features that were implemented throughout the code that contributed to making a secured execution.

1. Joint Angle Limits

During the inverse kinematics (IK) calculation in “solveIK” function (Figure 15), the joint angles are clamped to specified limits. This ensures that the arm does not exceed its physical constraints, preventing damage to the servos or structural components.

```
// Clamp joint angles within limits
float minAngle = -M_PI / 2;
float maxAngle = M_PI / 2;
if (i == 0) {
  minAngle = -M_PI;
  maxAngle = 0;
}

if (linkAnglesIK[i] < minAngle) {
  linkAnglesIK[i] = minAngle;
}
if (linkAnglesIK[i] > maxAngle) {
  linkAnglesIK[i] = maxAngle;
}
```

Figure 44. Angle Limitations Implementation

2. IK Iteration Limits

In the solveIK function (Figure 15), the algorithm limits the number of iterations to 100, thus if the solution does not converge within this limit, the process is aborted, and an error message is printed. This prevents the system from entering an infinite loop or performing unnecessary calculations.

```
Serial.println("IK did not converge within the iteration limit.");
```

Figure 45. IK Iteration Limit Message

3. Servo Validation

The setServoPos function (Figure 10) includes checks to ensure that the servo ID and the angle fall within valid ranges. This prevents invalid commands from being sent to the servos, protecting them from operating outside their physical or electrical limits.

4. Smooth Motion Control:

The incremental movement logic in the loop function ensures that servos move gradually toward their target positions, which avoids abrupt movements that could strain or damage the arm's structure or servos. This process was explained in detail in Servo and Speed Control Section.

5. Error Detection

The code includes error-handling mechanisms to detect missing or improperly connected components, such as the SD card or colour sensor. If an issue is detected, the program halts or skips operations to prevent further errors.

```
if (!SD.begin(SD_CS_PIN)) {
  Serial.println("SD Card initialization failed!");
  return;
}

if (!tcs.begin()) {
  Serial.println("No TCS34725 found ... check your connections");
  while (1); // Halt the program
}
```

Figure 46. Disconnected Components Error Messages

Challenges Faced

Hardware

When designing the hardware, it was noted that the SD card module used required 5V supply. The solution was to connect the VCC to VIN pin on the ESP32 which has direct connection to the USB 5V.

Firmware

Inverse Kinematics posed many challenges due to the limitations of visualising the robot. Also, compilation and upload time is considerable when performing iterative testing and debugging. To ease the development, cycle the IK algorithm was first tested in JS before it was ported to the C++. The algorithm used to solve the IK was finetuned to respect the limitations of the robot, but it has some drawbacks, there are positions that it fails to reach even when it is possible.

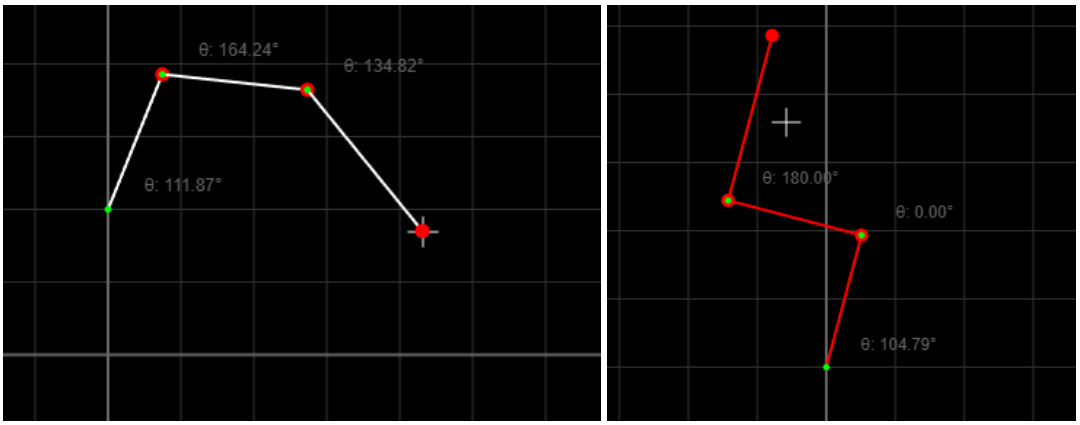


Figure 47. Visualization of the Inverse Kinematics Algorithm

Implementing Inverse Kinematics in 3D space can get very complex. But the robotic arm described in this report can be treated as 2D robot with rotating base. This is helpful when implementing IK. From the 3D coordinates we can derive the rotation where the robot should face, then we can solve the 2D inverse kinematics, and apply resulted rotations to the robot.

Web Interface

The biggest issue that was faced when developing the web interface was the ESP32 limitation to handle one network request at the same time. Modern servers are fast and capable of serving many clients and requests at the same time. When developing web interface with multiple files for the ESP32 it was crucial to make each request once the previous was finished. Additionally, it was common for the request to fail due to network related issues or since the ESP was not fast enough. To solve this problem, the web interface retries 3 times when the request fails. It also makes sure that there is only one request executed at any given time.

Inverse Kinematics Angles Mapping

During the implementation of the inverse kinematics in the “solveIK3D” function, we encountered a significant challenge with the default map function provided by the Arduino framework. This function is designed to scale input values from one range to another, but it operates exclusively with integers, which posed a problem because the calculations in the inverse kinematics process involve floating-point values, particularly when converting angles in radians to servo-compatible ranges.

To overcome this limitation, we implemented a custom “mymap” function, that unlike the default map function, it supports floating-point arithmetic, ensuring that the scaling of values is precise and free from rounding errors.

```
float mymap(float x, float in_min, float in_max, float out_min, float out_max) {
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Figure 48. Mymap Function

The function was specifically tailored to handle the conversion of joint angles from their calculated ranges to the servo's operational range of [0, 180].

V. Gripper Improvement

This section focuses on the enhancements made to the robotic arm's gripper, a critical component responsible for handling and manipulating objects. The primary objective of the gripper was to effectively grab and hold Skittles, which guided the development of each design. To explore diverse approaches and optimize performance, each team member designed a unique gripper tailored to this goal. By fostering individual creativity, the team aimed to identify innovative solutions and integrate the most effective features into the final design.

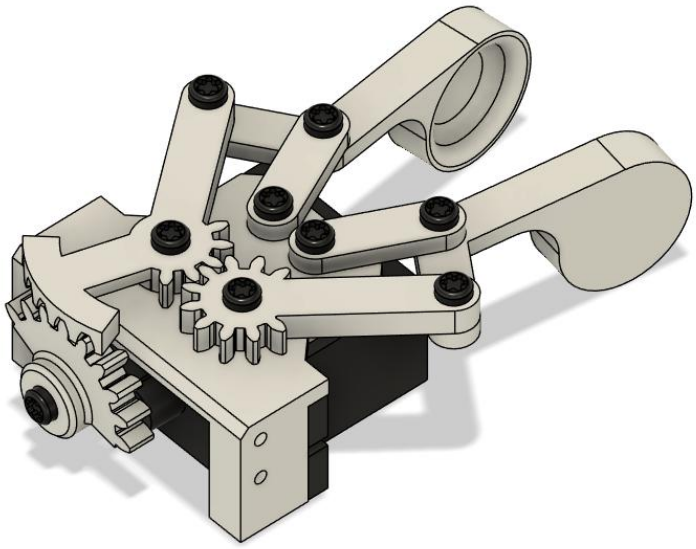


Figure 49. Gripper 3D Model

This gripper design was inspired by researching different gripper designs. It is complex, but the major benefit is the motor alignment; the motor is rotated 90deg to fit under most of the mechanical components to not protrude to the robot working area. The pinching mechanism was designed to allow for grabbing Skittles candies. After 3D printing, assembling and testing it was successful and determined suitable for the task.

VI. Testing and Results

Servo Angle Testing

Firstly, to validate the range of motion of the servo motors used in this project, we conducted a test to determine the maximum angles they could rotate. For this purpose, we designed and fabricated a custom testing device, as shown in the figure, which allowed us to visually measure the angles achieved by the servo and ensure that they aligned with the expected input commands.

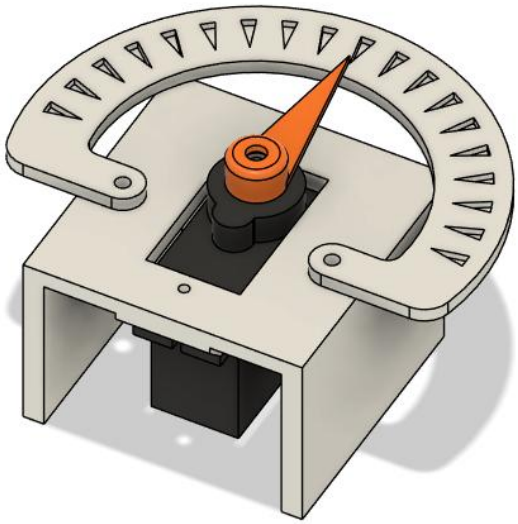


Figure 50. Servo Angle Testing Device

The device consists of three main components: a base with an integrated angle protractor, a pointer mounted on the servo's rotational axis, and the servo motor itself, secured within the base. The angle protractor was marked with degree measurements every 10 degrees, providing a clear visual reference for the servo’s rotation. The pointer, attached to the servo horn, moves across the protractor scale as the servo rotates, enabling precise measurements of the angles.

To perform the test, we programmed the servo to rotate to specific angles ranging from 0° to 180°, using the setServoPos function. As the servo moved, we recorded the pointer's position on the protractor to verify that the servo accurately reached the commanded angles. This process allowed us to confirm the operational range of the servos and detect any deviations or mechanical limitations that could impact the arm's performance.

Inverse and Forward Kinematics Testing

Secondly, to validate the consistency and accuracy of the robotic arm's inverse kinematics and forward kinematics calculations, we conducted a test to compare the input and output 3D coordinates (XYZ). The objective of this test was to ensure that the IK algorithms correctly calculated joint angles to reach a specified position and that the FK algorithms accurately reconstruct the resulting XYZ position based on the arm's final joint angles.

The testing procedure began by providing a set of target coordinates in 3D space through the robotic arm's API, which was tested using Postman. These coordinates were passed to the IK system, which calculated the joint angles necessary to position the arm's end-effector at the specified location. The calculated joint angles were then sent to the servos, moving the arm to the corresponding position. Once the robotic arm reached the target position, the FK system was used to calculate the actual XYZ coordinates of the end-effector based on the joint angles of the servos.

These reconstructed coordinates were compared to the original target XYZ coordinates to evaluate the accuracy of both the IK and FK systems. The process was repeated for multiple target positions to ensure consistency and reliability across the arm's workspace. This iterative testing process was critical for refining the arm's kinematic calculations and ensuring reliable operation for tasks requiring precise positioning.

Inverse Kinematics Demonstration Process

Thirdly and finally, to showcase the successful implementation of IK in the robotic arm, we developed a process that allows the user to send target coordinates directly to the robot via the web interface. This process used the G0 G-code command, which accepts XYZ coordinates and moves the robotic arm to the specified position in 3D space.

Figure 51. Example G0 Command for IK Demonstration

When a G0 command with target coordinates is sent through the web interface, as shown in the figure, the robotic arm begins by parsing the command using the “handleG0Command” function. This function extracts the XYZ coordinates and passes them to the “solveIK3D” function, which calculates the joint angles required to position the arm’s end-effector at the target location, the base rotation is determined, and finally the target is converted into a 2D plane for further calculations using the “solveIK function”. Once the angles are computed, the “setServoPos” function is used to map the calculated angles to servo positions, ensuring precise movement of the arm’s joints.

The robotic arm then moves to the specified position in real-time, allowing the user to visually confirm the accuracy of the IK implementation. This process not only validates the IK functionality but also provides a user-friendly way to control the arm for various applications, demonstrating its flexibility and responsiveness.

Sorting Process and Results

This whole project was designed with the objective of sorting Skittles based on their colour, utilizing a combination of custom hardware and software functions. The final setup consists of four key components: the robotic arm, the Skittle dropper, the base platform, the sorting cases, the electronic case and the robotic arm.

1. **Skittles Dropper**

It is composed of three main elements: a hopper, a sliding ramp, and a servo-controlled gate. The hopper holds the Skittles, allowing them to funnel down onto the ramp, the slide guides the Skittles toward the colour sensor station and the servo motor controls the gate, releasing one Skittle at a time to ensure accurate detection and processing.

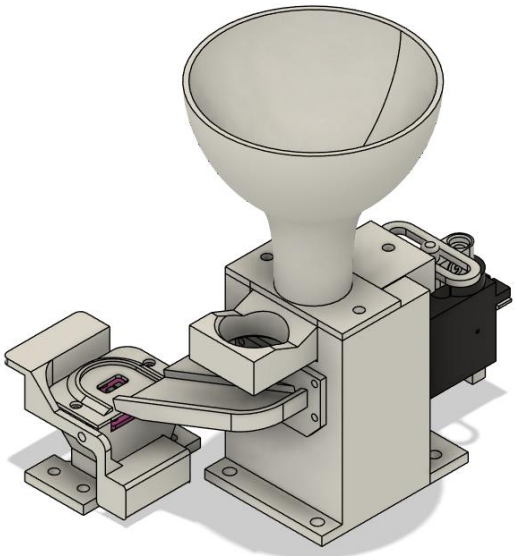


Figure 52. Skittles Dropper

2. Sorting Cases

These containers are designed to organize Skittles by colour after sorting, positioned at the base of the robotic arm's workspace, they provide a clear and accessible location for the robot to deposit them, ensuring easy separation and storage for each colour category.

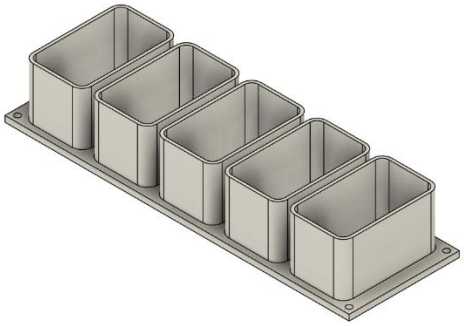


Figure 53. Sorting Cases

3. Base Platform

It serves as a stable foundation for the entire system, including the robotic arm, Skittles dropper, and sorting cases. It is perforated to allow easy attachment and alignment of all components, ensuring a secure and modular design.

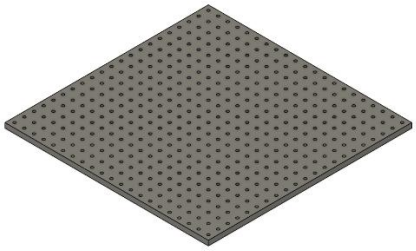


Figure 54. Base Platform

4. Electronic Case

The ESP32 microcontroller, PCA9685 servo driver, and SD card module are housed within the case, which is designed to neatly organize the electronic components. Separate power supplies are provided for the servo driver and ESP32 to ensure stable operation under heavy loads. Ventilation and cable management are incorporated into the enclosure to protect the electronics and maintain efficient wiring.

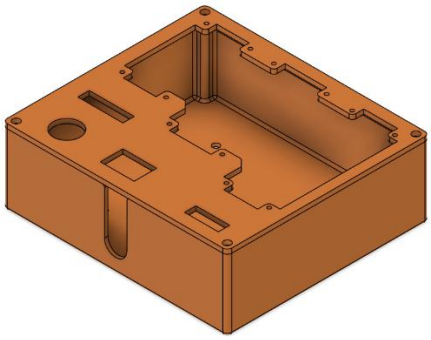


Figure 55. Electronic Case

5. Robotic Arm

The robotic arm is a multi-jointed assembly capable of precise movement to pick and place Skittles; features six servo motors controlled by the ESP32 and servo driver, enabling dynamic motion. The arm is equipped with a custom gripper for securely handling the objects during sorting tasks and its range of motion allows it to interact seamlessly with both the colour sensor station and the sorting cases.

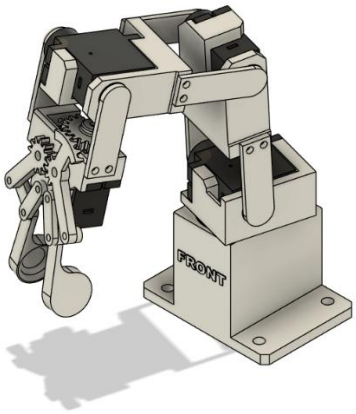


Figure 56. Robotic Arm

Except for the servo motors and screws used for assembly, all structural components of the system are entirely 3D-printed. This approach allowed for rapid prototyping, customization, and cost efficiency while maintaining functionality and durability. All these elements join to form the final setup, as the following figure shows.

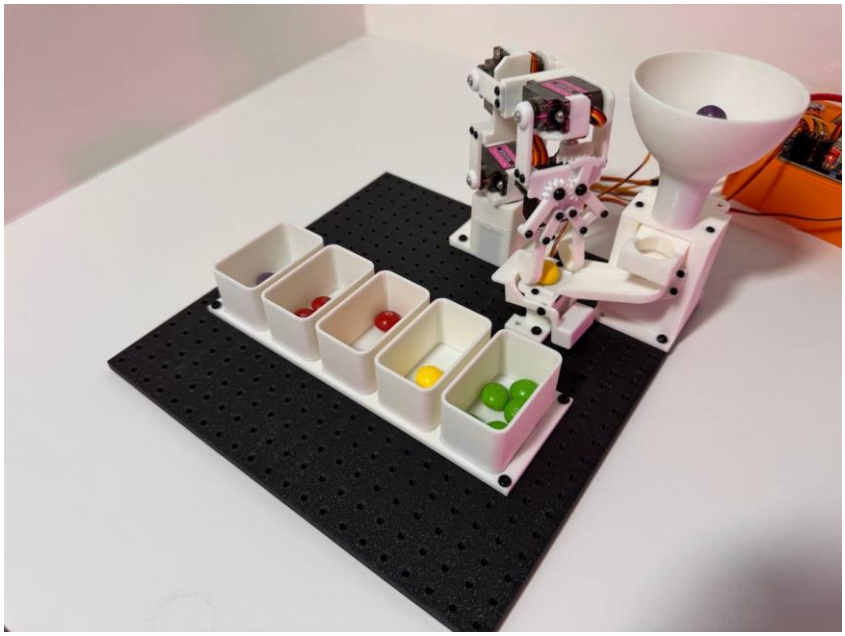


Figure 58. Final Setup of the Robotic System

Figure 57 Final Setup of the Robotic System

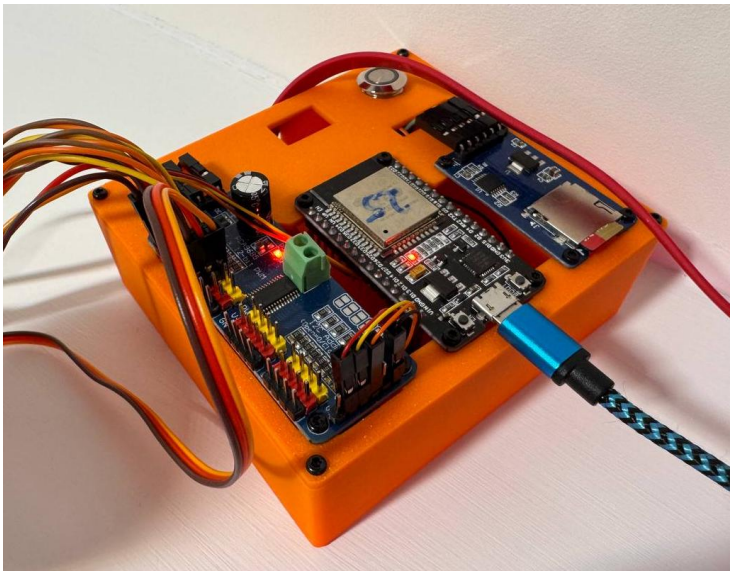


Figure 59. Circuit Box

The process begins with a Skittle sliding down the ramp from the dropper into the colour sensor stand, where its RGB values are measured using the colour sensor. The function “tcs.getRGB” retrieves the red, green, and blue components of the Skittle’s colour, and the “findClosestColorId” function compares these values to a predefined list of colours in the system (colorList), named in previous sections. Once the colour is identified, the robotic arm is instructed to act.

The robotic arm, controlled through forward kinematics, positions itself at the sensor stand to pick up the Skittle and, using the “setServoPos” function, the gripper opens to prepare for grabbing

the object. After closing the gripper to secure the object, the arm moves to the corresponding sorting case. The movement is coordinated through precise servo angle adjustments, which are pre-calculated for each sorting position. Upon reaching the target case, the gripper opens to release the Skittle into the appropriate container. The arm then returns to its starting position above the sensor stand to wait for the next object. This loop is repeated continuously, allowing the system to sort Skittles indefinitely until there are none left.

This process effectively demonstrates the robotic arm's ability to perform a complete pick-and-place task while integrating colour sorting functionality. By combining precise motion control, accurate colour detection, and seamless coordination between components, the system achieves the goal of sorting Skittles into their respective containers. The process can be observed in a video accessible through the following link: https://youtu.be/oouUzXQNhF4

VII. Conclusion

The 5-DoF robotic arm project successfully fulfilled its key objectives:

1. Structural Design and Optimization: The robotic arm’s structure, including a high-precision gripper, was effectively built and optimized using 3D-printed components and economical materials, able to pick and place an object.

2. Implementation of Kinematic Algorithms: Advanced forward and inverse kinematics were implemented, enabling precise motion control and positioning of the robotic arm.

3. Development of a Web-Based Interface: A user-friendly and interactive web interface was developed, providing seamless remote control and operation of the robotic system.

4. Integration of Wireless Connectivity: Wireless communication through the ESP32 microcontroller facilitated intuitive control and enhanced system accessibility.

These achievements underscore the project’s success in creating a versatile robotic arm suitable for precision-driven tasks like colour sorting. Safety mechanisms, robust error-handling features, and modular design further ensured the system’s reliability and adaptability.

Future improvements could include implementing object detection to avoid collisions within the workspace and designing a more refined and protective case for electronic components. These enhancements would further broaden the system's applications and usability, paving the way for industrial and research advancements in affordable robotics.

References

[1]. VD Cong, "Design and Development of a Cost efficiency Robot Arm with a PLC-based Robot Controller," *FME Transactions*, 2024. [Online]. Available: https://www.mas.bg.ac.rs/_media/istrazivanje/fme/vol52/2/6_v._d._cong.pdf.

[2]. J. Chang, "WPI Robotic Mascot," *Worcester Polytechnic Institute*, 2019. [Online]. Available: https://digital.wpi.edu/downloads/fn107154k.

[3]. L. Loughlin, "TRASHBOT: Mobile Garbage Collecting Robot," *Worcester Polytechnic Institute*, 2024. [Online]. Available: https://digital.wpi.edu/downloads/6h440x69q.

[4]. LFC Ccari, WA Aguilar, ES Espinoza, "Robust Finite-Time Adaptive Nonlinear Control System for an EOD Robotic Manipulator: Design, Implementation and Experimental Validation," *IEEE Transactions on Control Systems Technology*, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10587221/.

[5]. WA Salah, AA Sneineh, "Smartphone Sensor-based Development and Implementation of a Remotely Controlled Robot Arm," *Journal of Robotics and Control*, 2024. [Online]. Available: https://journal.umy.ac.id/index.php/jrc/article/view/21987.

[6]. M. Desai, H. Mewada, H. Chhajed, "Gesture Dependable Robotics ARM Movement: ROS and Machine Learning Perspective," in *Proceedings of the 9th South-East European Conference on Intelligent Systems*, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10734617/.

[7]. T. Saleh, "Three DoF articulated robotic manipulator for teaching and learning," *IET Conference Proceedings*, 2022. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/icp.2022.2264.

[8]. UU Deshpande, "Unmanned Drug Delivery Vehicle for COVID-19 Wards in Hospitals," *Journal of Computer Systems Research*, 2021. [Online]. Available: https://journals.bilpubgroup.com/index.php/jcsr/article/view/3489.

[9]. G. Kulkarni, A. Mahindrakar, "Kinematics And Structural Analysis Of 6 DOF Robotic Arm," *NeuroQuantology*, 2022. [Online]. Available: https://search.proquest.com/openview/cad9751f7ef017f06e3065be65f5afcd.

[10]. T. Wang, "Supporting the Design and Authoring of Pervasive Smart Environments," *Purdue University*, 2022. [Online]. Available: https://hammer.purdue.edu/articles/thesis/Supporting_the_Design_and_Authoring_of_Pervasive_Smart_Environments/19349831/1/files/34401371.pdf.

[11]. Autodesk, Inc., "Fusion 360," Autodesk, Inc., San Rafael, CA, USA. [Online]. Available: https://www.autodesk.com/products/fusion-360

[12]. Espressif Systems, "ESP32 Series Datasheet," 2024. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[13]. Components101, "MG90S Micro Servo Motor Datasheet, Wiring Diagram & Features," n.d. [Online]. Available: https://components101.com/motors/mg90s-metal-gear-servo-motor.

[14]. ams AG, "TCS3472 Datasheet: Color Light-to-Digital Converter with IR Filter," 2016. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/TCS34725.pdf.

[15]. NXP Semiconductors, "PCA9685 16-Channel, 12-bit PWM Fm+ I²C-Bus LED Controller Datasheet," Apr. 2015. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/PCA9685.pdf.

Appendices

Main Code

```
#include <WiFi.h>
#include <FS.h>
#include <SD.h>
#include <WebServer.h>
#include <ArduinoJson.h>
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>
#include "Adafruit_TCS34725.h"
#include <math.h>

#define LED_PIN 2 // Define the GPIO for the LED
#define SD_CS_PIN 5 // SD card CS pin (adjust according to your wiring)

struct ColorEntry {
  float r, g, b;
};

struct Position {
  float x;
  float y;
};

// Link lengths (IK)
float linkLengths[] = {50.0, 50.0, 75.0};
float linkAnglesIK[] = {-1.57, 0, 1.57};
const int numLinks = 3;
const float tolerance = 0.5;

ColorEntry colorList[] = {
  ColorEntry{93.25, 92.5, 62.5}, // Empty (no object detected)
  ColorEntry{111.5, 86.5, 62},   // Purple
  ColorEntry{150, 69.5, 50.5},   // Red
  ColorEntry{144, 70, 42},       // Orange
  ColorEntry{122, 90, 40.5},     // Yellow
  ColorEntry{93.5, 111.25, 51}   // Green
};

int numColors = 6;

// WiFi credentials
const char* ssid = "WIFI SSID";
const char* password = "PASSWORD";

// Create a web server on port 80
```

```
WebServer server(80);

#define SERVOMIN  80 // This is the 'minimum' pulse length count (out of 4096)
#define SERVOMAX  460 // This is the 'maximum' pulse length count (out of 4096)
#define SERVOMID  270
#define USMIN  600 // This is the rounded 'minimum' microsecond length based on the minimum
pulse of 150
#define USMAX  2400 // This is the rounded 'maximum' microsecond length based on the maximum
pulse of 600
#define SERVO_FREQ 50 // Analog servos run at ~50 Hz updates
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x40);
Adafruit_TCS34725 tcs = Adafruit_TCS34725(TCS34725_INTEGRATIONTIME_2_4MS, TCS34725_GAIN_4X);

int servoPos[16];
int servoPosGoal[16];
bool busy = false;

long timeToUpdateServo = 0;

String inputString = "";    // A string to hold incoming data
bool stringComplete = false; // Whether the string is ready to be processed

float sns_red, sns_green, sns_blue;
int color_id = -1;

//G-code
int gCodePointer = 0;
long blockExecutionUntil = 0;
bool runGCodeExecution = false; //runGCodeProgram
int ifStack = 0;

String program[] = {
  ":START",
  "G1 A90 B90 C60 D20 E65 F0",
  ":DISPENSE",
  "G1 A150 B90 C10 D5 E65",
  "G1 F180",
  "WAIT 500",
  "G1 F0",
  "WAIT 500",
  ":GET_COLOR",
  "COLOR",
  "IF COLOR == 0",
  "WAIT 2000",
  "JMP DISPENSE",
  "END",
  ":GRAB",
  "G1 E90",
  "WAIT 500",
  "G1 A150 B100 C30 D5",
```

```
  "// DROP AT PURPLE BOX",
  "IF COLOR == 1",
  "G1 A60 B70 C30 D50",
  "G1 E70",
  "END",
  "// DROP AT RED BOX",
  "IF COLOR == 2",
  "G1 A75 B80 C30 D30",
  "G1 E70",
  "END",
  "// DROP AT ORANGE BOX",
  "IF COLOR == 3",
  "G1 A90 B80 C30 D30",
  "G1 E70",
  "END",
  "// DROP AT YELLOW BOX",
  "IF COLOR == 4",
  "G1 A110 B80 C30 D30",
  "G1 E70",
  "END",
  "// DROP AT GREEN BOX",
  "IF COLOR == 5",
  "G1 A125 B55 C60 D20",
  "G1 E70",
  "END",
  "G1 E70",
  "WAIT 1000",
  "G1 A90 B90 C60 D20 E70",
  "WAIT 250",
  "JMP START"
};

void logRequest() {
  String clientIP = server.client().remoteIP().toString(); // Get client IP
  String method = (server.method() == HTTP_GET) ? "GET" : "POST"; // Determine request method
  String path = server.uri(); // Requested path

  // Log client IP, method, and requested path
  Serial.print("[");
  Serial.print(millis());
  Serial.print("] ");
  Serial.print("Client ");
  Serial.print(clientIP);
  Serial.print(" requested ");
  Serial.print(method);
  Serial.print(" ");
  Serial.println(path);
}

void handlePreflight() {
```

```
    // Send CORS headers for preflight response
    handleCORS();
    server.send(204); // No Content response for preflight
}

void handleCORS() {
    server.sendHeader("Access-Control-Allow-Origin", "*");
    server.sendHeader("Access-Control-Allow-Methods", "GET, POST, OPTIONS");
    server.sendHeader("Access-Control-Allow-Headers", "Content-Type");
}

// Function to serve static files from SD card
void handleFileRequest() {
  logRequest();
  if (server.method() == HTTP_OPTIONS) {
    handlePreflight();
    return;
  }

  String path = server.uri();
  if (path.endsWith("/")) {
    path += "index.html"; // Serve index.html if root directory is accessed
  }

  String contentType = getContentType(path);

  // Try to open the requested file
  if (SD.exists(path)) {
    File file = SD.open(path);
    Serial.print("File ");
    Serial.print(path);
    Serial.println(" found!");
    server.streamFile(file, contentType);
    file.close();
  } else {
    // If file is not found, return 404
    server.send(404, "text/plain", "File Not Found");
  }
}

// Helper function to determine content type
String getContentType(String filename) {
  if (filename.endsWith(".html")) return "text/html";
  if (filename.endsWith(".css")) return "text/css";
  if (filename.endsWith(".js")) return "application/javascript";
  if (filename.endsWith(".png")) return "image/png";
  if (filename.endsWith(".jpg")) return "image/jpeg";
  if (filename.endsWith(".gif")) return "image/gif";
  if (filename.endsWith(".json")) return "application/json";
  return "text/plain";
```

```
}

// API handler for toggling LED
void handleToggleLED() {
  logRequest();
  static bool ledState = false;
  ledState = !ledState;
  digitalWrite(LED_PIN, ledState ? HIGH : LOW);
  server.send(200, "application/json", "{\"status\":\"LED toggled\"}");
}

// API handler for returning millis()
void handleGetMillis() {
  logRequest();
  unsigned long uptime = millis();
  String jsonResponse = "{\"millis\":" + String(uptime) + "}";
  server.send(200, "application/json", jsonResponse);
}


// API handler for RGB sensor
void handleGetRGB() {
  logRequest();
  String jsonResponse = "{\"R\":" + String(sns_red) + ",\"G\":" + String(sns_green) + ",\"B\":"
+ String(sns_blue) + "}";
  server.send(200, "application/json", jsonResponse);
}


// API handler for setting multiple servo positions via POST request
void setServoPosition() {
  logRequest();
  handleCORS();

  // Check if request body contains JSON
  if (server.hasArg("plain")) {
    // Parse the JSON
    StaticJsonDocument<400> doc;
    DeserializationError error = deserializeJson(doc, server.arg("plain"));

    // Handle JSON parsing error
    if (error) {
      server.send(400, "application/json", "{\"status\":\"Invalid JSON\"}");
      return;
    }

    // Create a response object
    DynamicJsonDocument response(200);

    // Loop over each key-value pair in the JSON object
```

```
    for (JsonPair kv : doc.as<JsonObject>()) {
      String key = kv.key().c_str();

      // Check if key is in the format "servoX" where X is an integer
      if (key.startsWith("servo") && key.length() > 5) {
        int servoID = key.substring(5).toInt(); // Extract servo ID number
        int position = kv.value().as<int>();    // Get servo position

        // Ensure the position is within valid bounds (0-180)
        if (position >= 0 && position <= 180) {
          setServoPos(servoID, position);
          response[key] = "Updated to " + String(position);
        } else {
          response[key] = "Position out of range";
        }
      } else {
        response[key] = "Invalid key format";
      }
    }

    // Send response with the status for each servo updated
    String jsonResponse;
    serializeJson(response, jsonResponse);
    server.send(200, "application/json", jsonResponse);
  } else {
    // Respond with an error if no body is provided
    server.send(400, "application/json", "{\"status\":\"No JSON body found\"}");
  }
}

void getServoPosition() {
  //logRequest();
  handleCORS();

  // Create a JSON document to store the response
  DynamicJsonDocument response(400); // Adjust size if you have more than 16 servos

  // Map raw PWM values to angles (0-180 degrees) for each servo
  for (int i = 0; i < 16; i++) { // Assuming 16 servos
    int angle = map(servoPosGoal[i], SERVOMIN, SERVOMAX, 0, 180); // Map to 0-180
    String servoKey = "servo" + String(i); // Create key in the format "servoX"
    response[servoKey] = angle;
  }

  // Serialize the response to JSON
  String jsonResponse;
  serializeJson(response, jsonResponse);

  // Send the JSON response
  server.send(200, "application/json", jsonResponse);
```

```
}


// API handler for setting multiple servo positions via POST request
void setServoPWM() {
  logRequest();

  if (server.method() == HTTP_OPTIONS) {
    handleCORS();
    server.send(204); // No Content
    return;
  }

  // Check if request body contains JSON
  if (server.hasArg("plain")) {
    // Parse the JSON
    StaticJsonDocument<400> doc;
    DeserializationError error = deserializeJson(doc, server.arg("plain"));

    // Handle JSON parsing error
    if (error) {
      server.send(400, "application/json", "{\"status\":\"Invalid JSON\"}");
      return;
    }

    // Create a response object
    DynamicJsonDocument response(200);

    // Loop over each key-value pair in the JSON object
    for (JsonPair kv : doc.as<JsonObject>()) {
      String key = kv.key().c_str();

      // Check if key is in the format "servoX" where X is an integer
      if (key.startsWith("servo") && key.length() > 5) {
        int servoID = key.substring(5).toInt(); // Extract servo ID number
        int position = kv.value().as<int>();    // Get servo position

        // Ensure the position is within valid bounds (0-180)
        if (position >= 0 && position <= 4096) {
          //pwm.setPWM(i, 0, position);
          servoPosGoal[servoID] = position;
          //servoPos[i] = position;

          //setServoPos(servoID, position);
          response[key] = "Updated to " + String(position);
        } else {
          response[key] = "Position out of range";
        }
      } else {
        response[key] = "Invalid key format";
```

```
      }
    }

    // Send response with the status for each servo updated
    String jsonResponse;
    serializeJson(response, jsonResponse);
    server.send(200, "application/json", jsonResponse);
  } else {
    // Respond with an error if no body is provided
    server.send(400, "application/json", "{\"status\":\"No JSON body found\"}");
  }
}

// API handler for running a GCode command
void runGCodeCommandHandler() {
  logRequest();
  handleCORS();

  // Check if request body contains JSON
  if (server.hasArg("plain")) {
    // Parse the JSON
    StaticJsonDocument<200> doc;
    DeserializationError error = deserializeJson(doc, server.arg("plain"));

    // Handle JSON parsing error
    if (error) {
      server.send(400, "application/json", "{\"status\":\"Invalid JSON\"}");
      return;
    }

    // Extract the "command" field from the JSON
    if (doc.containsKey("command")) {
      String command = doc["command"].as<String>();

      // Run the GCode command
      runGCodeCommand(command, false);

      // Respond with success
      String response = "{\"status\":\"Command executed successfully\", \"command\":\"" +
command + "\"}";
      server.send(200, "application/json", response);
    } else {
      // Respond with error if "command" key is missing
      server.send(400, "application/json", "{\"status\":\"Missing 'command' field\"}");
    }
  } else {
    // Respond with an error if no body is provided
    server.send(400, "application/json", "{\"status\":\"No JSON body found\"}");
  }
}
```

```
// Validate and Update setServoPos() function
void setServoPos(uint8_t n, float angle) {
  // Validate if the servo ID is within range (0-15)
  if (n < 0 || n >= 16) {
    Serial.println("Invalid servo ID");
    return;
  }

  if(angle < 0 || angle > 180){
    Serial.println("Invalid servo angle");
    return;
  }

  // Map the angle to the appropriate PWM value
  uint16_t pwmVal = map(angle, 0, 180, SERVOMIN, SERVOMAX);
  //pwm.setPWM(n, 0, pwmVal);  // Corrected to send PWM on channel 'n'
  servoPosGoal[n] = pwmVal;
  if(n == 5){
    // servoPos[n] = pwmVal;
    if(pwmVal > 90){
      pwmVal = 90;
    }
  }

  Serial.print("Servo ");
  Serial.print(n);
  Serial.print(" set to: ");
  Serial.print(angle);
  Serial.print("deg (");
  Serial.print(pwmVal);
  Serial.println(")");

}

void setServoPulse(uint8_t n, double pulse){
  double pulselength;

  pulselength = 1000000;   // 1,000,000 us per second
  pulselength /= SERVO_FREQ;   // Analog servos run at ~60 Hz updates
  Serial.print(pulselength); Serial.println(" us per period");
  pulselength /= 4096;  // 12 bits of resolution
  Serial.print(pulselength); Serial.println(" us per bit");
  pulse *= 1000000;  // convert input seconds to us
  pulse /= pulselength;
  Serial.println(pulse);
  pwm.setPWM(n, 0, pulse);
}
```

```
void setup() {
  Serial.begin(115200);
  Serial.println("Serial begin");

  // Initialize LED pin
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW); // Start with LED off

  Serial.println("TCS34725 Init");
  if (tcs.begin()) {
    Serial.println("Found sensor");
  } else {
    Serial.println("No TCS34725 found ... check your connections");
    while (1); // halt!
  }

  tcs.setInterrupt(true); //turn off LED

  Wire.begin();
  pwm.begin();
  pwm.setOscillatorFrequency(27000000);
  pwm.setPWMFreq(50);
  Wire.setClock(100000);

  for(int i=0; i<16; i++){
    pwm.setPWM(i, 0, SERVOMID);
    servoPosGoal[i] = SERVOMID;
    servoPos[i] = SERVOMID;
  }

  // Initialize SD card
  if (!SD.begin(SD_CS_PIN)) {
    Serial.println("SD Card initialization failed!");
    return;
  }else{
    Serial.println("SD Card initialized.");
  }

  // Connect to WiFi
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected. IP address: ");
  Serial.println(WiFi.localIP());

  // Define API routes
  server.on("/api/toggle-led", HTTP_GET, handleToggleLED); // Endpoint to toggle LED
```

```
  server.on("/api/millis", HTTP_GET, handleGetMillis); // Endpoint to get uptime in millis
  server.on("/api/rgb", HTTP_GET, handleGetRGB);
  server.on("/api/servo", HTTP_POST, setServoPosition);
  server.on("/api/servo", HTTP_GET, getServoPosition);
  server.on("/api/set-pwm", HTTP_POST, setServoPWM);
  server.on("/api/gcode", HTTP_POST, runGCodeCommandHandler);
  server.on("/api/gcode", HTTP_OPTIONS, handlePreflight);

  // Route to serve files from the SD card
  server.onNotFound(handleFileRequest); // Handle all other requests as file requests

  // Start the web server
  server.begin();
  Serial.println("HTTP server started.");
}

void loop() {
  //unsigned long timeMs = millis();

  //must be first
  if(timeToUpdateServo < millis()){
    if(millis() - timeToUpdateServo > 25){ // show warning when loop is behind
      Serial.print("Warning, servo loop behind ");
      Serial.print(millis() - timeToUpdateServo);
      Serial.println("ms");
    }

    timeToUpdateServo = millis() + 16;

    bool reachedGoad = true;
    for(int i=0; i<16; i++){
      if(abs(servoPos[i] - servoPosGoal[i]) < 3){
        servoPos[i] = servoPosGoal[i];
      }else{
        if(servoPos[i] < servoPosGoal[i]){
          servoPos[i]+=2;
        }

        if(servoPos[i] > servoPosGoal[i]){
          servoPos[i]-=2;
        }
      }

      //check if all motors reached the goal
      if(servoPos[i] != servoPosGoal[i]){
        reachedGoad = false;
      }

      pwm.setPWM(i, 0, servoPos[i]);
    }
```

```
    busy = !reachedGoad;
    digitalWrite(LED_PIN, busy);
  }

  // if(timeToGetColor < timeMs && !(timeToUpdateServo < timeMs-5)){
  //   if(!readingInProgress){
  //     tcs.setInterrupt(false); // Turn on LED, start the reading
  //     readingInProgress = true;
  //     timeToGetColor = timeMs + 5;
  //   }else{
  //     tcs.getRGB(&sns_red, &sns_green, &sns_blue);
  //     tcs.setInterrupt(true); // Turn off LED
  //     readingInProgress = false;
  //     timeToGetColor = timeMs + 995;

  //     int cid = findClosestColorId(sns_red, sns_green, sns_blue);
  //     if(cid == 1){Serial.println("PURPLE");}else
  //     if(cid == 2){Serial.println("RED");}else
  //     if(cid == 3){Serial.println("ORANGE");}else
  //     if(cid == 4){Serial.println("YELLOW");}else
  //     if(cid == 5){Serial.println("GREEN");}

  //   }
  // }

  //execute Serial commands
  if (stringComplete) {
    runGCodeCommand(inputString, false);  // Process the command
    inputString = "";             // Clear the string for the next input
    stringComplete = false;       // Reset the flag
  }

  runGCodeProgram();

  // Handle client requests
  server.handleClient();
}
// G-Code ---------------------------------------------------
void runGCodeProgram(){
  if(!busy && blockExecutionUntil < millis() && runGCodeExecution){

    if (gCodePointer >= sizeof(program) / sizeof(program[0])) {
      runGCodeCommand("STOP", false);
      return;
    }

    //trip leading spaces
    // int i=0;
    // while (i < program.length() && program.charAt(gCodePointer) == ' ') {
    //   i++;
```

```
    // }
    String line = program[gCodePointer];
    line.trim();
    //String line = program[gCodePointer].substring(i);

    if(isLineExecutable(line)){
      Serial.print("[G-Code] [");
      Serial.print(line);
      Serial.print("]: ");
      runGCodeCommand(line, true);
    }else{
      gCodePointer++;
    }
  }
}

bool isLineExecutable(const String& line) {
  if (line.charAt(0) == ':' || line.charAt(0) == '/' || line.length() == 0 || line.charAt(0) ==
' ') {
    return false;
  }
  return true;
}


// Executes G-Code command
void runGCodeCommand(const String& command, bool advanceProgram) {
    // Find the first space to separate the command from the parameters
    int spaceIndex = command.indexOf(' ');
    String gcodeCommand;
    String parameters;

    if (spaceIndex != -1) {
        // Extract the command and parameters
        gcodeCommand = command.substring(0, spaceIndex);
        parameters = command.substring(spaceIndex + 1);
    } else {
        // No parameters, the entire string is the command
        gcodeCommand = command;
        parameters = "";
    }

    // Call the appropriate handler based on the command
    if (gcodeCommand.equals("G0")) {
        Serial.println(command);
        handleG0Command(parameters);
        if (advanceProgram) gCodePointer++;
    }
    else
    if (gcodeCommand.equals("G1")) {
```

```
        Serial.println(command);
        handleG1Command(parameters);
        if (advanceProgram) gCodePointer++;
    }
    else if (gcodeCommand.equals("COLOR")) {
        tcs.setInterrupt(false); // Turn on LED
        delay(10);
        tcs.getRGB(&sns_red, &sns_green, &sns_blue);
        tcs.setInterrupt(true); // Turn off LED
        Serial.print("COLOR (");
        Serial.print(sns_red);
        Serial.print(", ");
        Serial.print(sns_green);
        Serial.print(", ");
        Serial.print(sns_blue);
        Serial.print(") is ID: ");
        color_id = findClosestColorId(sns_red, sns_green, sns_blue);
        Serial.println(color_id);
        if (advanceProgram) gCodePointer++;
    }
    else if(gcodeCommand.equals("JMP")){
      jumpToLabel(":"+parameters);
    }
    else if(gcodeCommand.equals("WAIT")){
      int waitTime = parameters.toInt();
      blockExecutionUntil = millis() + waitTime;
      Serial.print("Waiting ");
      Serial.print(waitTime);
      Serial.println("ms");
      if (advanceProgram) gCodePointer++;
    }
    else if(gcodeCommand.equals("START")){
      runGCodeExecution = true;
      Serial.println("STARTING THE PROGRAM");
      if (advanceProgram) gCodePointer++;
    }
    else if(gcodeCommand.equals("STOP")){
      runGCodeExecution = false;
      gCodePointer = 0;
      Serial.println("STOPPING THE PROGRAM");
    }
    else if(gcodeCommand.equals("PAUSE")){
      runGCodeExecution = false;
      Serial.println("PAUSING THE PROGRAM");
    }
    else if(gcodeCommand.equals("IF")){
      handleIfCondition(parameters);
      if (advanceProgram) gCodePointer++;
    }
    else if(gcodeCommand.equals("END")){
```

```
      handleEndCondition();
      if (advanceProgram) gCodePointer++;
    }
    else {
        // Unknown command, handle error or ignore
        Serial.println("Unknown command: " + gcodeCommand);
    }
}

void handleG0Command(const String& command) {
    // Map motor labels to motor numbers
    const uint8_t motorMap[] = {'X', 'Y', 'Z', 'E'};
    float x = 0;
    float y = 0;
    float z = 0;
    float e = 0;
    // Loop through each motor label
    for (uint8_t i = 0; i < 4; i++) {
        bool found = false;
        double val = extractValue(command, motorMap[i], found);
        if (found) {
            if(motorMap[i] == 'X') x = val;
            if(motorMap[i] == 'Y') y = val;
            if(motorMap[i] == 'Z') z = val;
            if(motorMap[i] == 'E') e = val;
        }
    }
    Serial.println("----");
    Serial.print("Solving IK for position ");
    Serial.print("X:");
    Serial.print(x);
    Serial.print(", Y:");
    Serial.print(y);
    Serial.print(", Z:");
    Serial.println(z);

    solveIK3D(x, y, z);

    Position p = calculatePosition(numLinks);
    Serial.print("Reached to X:");
    Serial.print(p.x);
    Serial.print(", Y:");
    Serial.println(p.y);
}

void handleG1Command(const String& command) {
    // Map motor labels to motor numbers
    const uint8_t motorMap[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P'};
```

```
    // Loop through each motor label
    for (uint8_t i = 0; i < 16; i++) {
        bool found = false;
        double angle = extractValue(command, motorMap[i], found);
        if (found) {
            // Set the servo position for the specific motor
            setServoPos(i, angle);
        }
    }
}

// Function to extract a value for a specific parameter (e.g., 'A', 'B', etc.)
double extractValue(const String& command, char param, bool& found) {
    int index = command.indexOf(param);
    if (index != -1) {
        int endIndex = command.indexOf(' ', index + 1); // Find end of the value
        if (endIndex == -1) endIndex = command.length(); // Last parameter
        String value = command.substring(index + 1, endIndex);
        found = true;
        return value.toDouble(); // Convert to double
    }
    found = false;
    return 0.0;
}

void jumpToLabel(const String& label) {
  for (int i = 0; i < sizeof(program) / sizeof(program[0]); i++) {
    if (program[i].equals(label)) {
      gCodePointer = i + 1;
      Serial.print("Jumping to '");
      Serial.print(label);
      Serial.print("'at line: ");
      Serial.println(gCodePointer);
      break;
    }
  }
}

//EXAMPLE
//COLOR == 1
void handleIfCondition(const String& parameters) {
    // Increment the IF stack
    ifStack++;

    // Find the operator ("==", "!=") and split the condition and value
    int operatorIndex1 = parameters.indexOf("==");
    int operatorIndex2 = parameters.indexOf("!=");
    bool conditionResult = false;

    if (operatorIndex1 == -1 && operatorIndex2 == -1) {
```

```
      Serial.print("Invalid condition: ");
      return;
    }

    String condition = parameters.substring(0, max(operatorIndex1, operatorIndex2));
    condition.trim();

    if(operatorIndex1 > 0){
      String value = parameters.substring(operatorIndex1 + 2);
      value.trim();
      int targetValue = value.toInt();
      if(targetValue == color_id){
        conditionResult = true;
      }
    }

    if(operatorIndex2 > 0){
      String value = parameters.substring(operatorIndex2 + 2);
      value.trim();
      int targetValue = value.toInt();
      if(targetValue == color_id){
        conditionResult = true;
      }
    }

    // Log the evaluation result
    if (conditionResult) {
        Serial.print("IF CONDITION '");
        Serial.print(parameters);
        Serial.println("' IS TRUE!");
    } else {
        Serial.print("IF CONDITION '");
        Serial.print(parameters);
        Serial.println("' IS FALSE!");
        skipUntilEnd(); // Skip to the next END block
    }
}

void handleEndCondition() {
  Serial.println("IF END");
  ifStack--;
  if (ifStack < 0) {
    Serial.println("Unmatched END in G-code");
    runGCodeExecution = false;
  }
}

void skipUntilEnd() {
  while (gCodePointer < sizeof(program) / sizeof(program[0])) {
    if (program[gCodePointer].equals("END")) {
```

```
      gCodePointer++;
      break;
    }
    gCodePointer++;
  }
}

// This function is called when serial data is received
void serialEvent() {
  while (Serial.available()) {
    char inChar = (char)Serial.read(); // Read the incoming byte
    // If a newline character is detected, set the flag
    if (inChar == '\n') {
      stringComplete = true;
    }else{
      inputString += inChar;
    }
  }
}

int findClosestColorId(float r, float g, float b) {
  float minDistance = 999999.0;  // Initialize to a large number
  int closestId = 0;  // Default to no match

  for (int i = 0; i < numColors; i++) {
    // Calculate the Euclidean distance between the input color and each color in the list
    float distance = sqrt(pow(r - colorList[i].r, 2) + pow(g - colorList[i].g, 2) + pow(b -
colorList[i].b, 2));

    // Check if the distance is the smallest so far and within a reasonable threshold
    if (distance < minDistance && distance < 15) {
      minDistance = distance;
      closestId = i;  // Store the index of the closest color
    }
  }

  return closestId;
}

void solveIK3D(float x, float y, float z){
  y = y-50; //move target by 50mm for 2D IK
  float baseRot = atan2(z, x);
  float distToY = sqrt(x * x + z * z);
  solveIK(distToY, -y);

  Serial.print("2D IK space: (");
  Serial.print(distToY);
  Serial.print(", ");
  Serial.print(y);
  Serial.println(")");
```

```
  float br = mymap(baseRot, 0, M_PI, 180, 0);
  Serial.print("Base rot: ");
  Serial.print(baseRot);
  Serial.print("rad, ");
  Serial.print(br);
  Serial.println("deg");

  // Print the joint angles
  for (int i = 0; i < numLinks; i++) {
    Serial.print("Angle ");
    Serial.print(i + 1);
    Serial.print(": ");
    Serial.print(linkAnglesIK[i]);
    Serial.print("rad, ");
    if(i==0){
      Serial.print(mymap(linkAnglesIK[i], -M_PI, 0, 0, 180)); //conver to deg
    }else{
      Serial.print(mymap(linkAnglesIK[i], -M_PI/2, M_PI/2, 0, 180)); //conver to deg
    }
    Serial.println("deg");
  }

  setServoPos(0, br);
  setServoPos(1, mymap(linkAnglesIK[0], -M_PI, 0, 180, 0));
  setServoPos(2, mymap(linkAnglesIK[1], -M_PI/2, M_PI/2, 180, 0));
  setServoPos(3, mymap(linkAnglesIK[2], -M_PI/2, M_PI/2, 180, 0));
}

// Function to calculate the position of the end of the chain
Position calculatePosition(int iter) {
  float x = 0;
  float y = 0;
  float angleSum = 0;

  for (int i = 0; i < iter && i < numLinks; i++) {
    angleSum += linkAnglesIK[i];
    x += linkLengths[i] * cos(angleSum);
    y += linkLengths[i] * sin(angleSum);
  }

  Position result = {x, y};
  return result;
}

// Cyclic Coordinate Descent (CCD) implementation
void solveIK(float targetX, float targetY) {
  linkAnglesIK[0] = -1.57;
  linkAnglesIK[1] = 0;
  linkAnglesIK[2] = 1.57;
```

```
  for (int iter = 0; iter < 100; iter++) {
    for (int i = numLinks - 1; i >= 0; i--) {
      Position currentPos = calculatePosition(numLinks); // End effector position
      float distToTarget = sqrt(pow(currentPos.x - targetX, 2) + pow(currentPos.y - targetY,
2));

      // Check if within tolerance
      if (distToTarget < tolerance) {
        Serial.println("Target reached.");
        return;
      }

      Position jointPos = calculatePosition(i); // Position of the current joint

      // Vectors
      float jointToEffectorX = currentPos.x - jointPos.x;
      float jointToEffectorY = currentPos.y - jointPos.y;
      float jointToTargetX = targetX - jointPos.x;
      float jointToTargetY = targetY - jointPos.y;

      // Normalize vectors
      float effectorLength = sqrt(jointToEffectorX * jointToEffectorX + jointToEffectorY *
jointToEffectorY);
      float targetLength = sqrt(jointToTargetX * jointToTargetX + jointToTargetY *
jointToTargetY);

      jointToEffectorX /= effectorLength;
      jointToEffectorY /= effectorLength;
      jointToTargetX /= targetLength;
      jointToTargetY /= targetLength;

      // Compute angle between vectors
      float dot = jointToEffectorX * jointToTargetX + jointToEffectorY * jointToTargetY;
      float det = jointToEffectorX * jointToTargetY - jointToEffectorY * jointToTargetX;
      float angleChange = atan2(det, dot);

      // Update joint angle
      linkAnglesIK[i] += angleChange;

      // Clamp joint angles within limits
      float minAngle = -M_PI / 2;
      float maxAngle = M_PI / 2;
      if (i == 0) {
        minAngle = -M_PI;
        maxAngle = 0;
      }

      if (linkAnglesIK[i] < minAngle) {
        linkAnglesIK[i] = minAngle;
      }
```

```
      if (linkAnglesIK[i] > maxAngle) {
        linkAnglesIK[i] = maxAngle;
      }
    }
  }

  Serial.println("IK did not converge within the iteration limit.");
}

float mymap(float x, float in_min, float in_max, float out_min, float out_max) {
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Web Interface Code

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Robot</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.4.0/p5.js"></script>
    <style>
        * {
            font-family: Roboto-Regular, Arial, Helvetica, sans-serif;
            color: #fff;
            background-color: #000000;
        }

        #canvas-container {
            display: flex;
            justify-content: center;
            align-items: center;
            height: 500px;
            width: 800px;
            border: 1px solid rgb(53, 53, 53);
            margin-bottom: 20px;
```

```
}

#gui {
    display: flex;
    justify-content: center;
    align-items: center;
    flex-direction: column;
}

#dark {
    color: #6e6e6e;
}

#gui span {
    color: #6e6e6e;
}

#gui label {
    color: #ffffff;
}

#slider-container {
    display: flex;
    justify-content: center;
    align-items: left;
    width: 800px;
    flex-direction: column;
}

#buttons-container {
    display: flex;
    gap: 5px;
    align-items: center;
    flex-direction: row;
}

#heart-beat {
    width: 25px;
```

```
            height: 25px;
            border-radius: 50%;
            background-color: #55dd16;
        }

        input[type=range] {
            width: 100%;
            margin: 5px 0;
        }

        input[type=checkbox] {
            margin: 5px 0;
        }

        input[type=text] {
            width: 400px;
            margin: 5px 0;
            font-size: 20px;
            color: #dadada;
            background-color: #1b1b1b;
            border: 1px solid #8f8f8f;
        }

        button {
            width: 100px;
            margin: 5px 0;
            font-size: 20px;
            color: #ffffff;
            background-color: #1b1b1b;
            border: 1px solid #8f8f8f;
            cursor: pointer;
        }
    </style>
</head>
<body>
    <main>
        <div id="gui">
            <h1>Robot arm control panel</h1>
```

```
            <div id="canvas-container"></div> <!-- Canvas will go here --
>
            <div id="slider-container">
                <span id="dark">Viewport: <span id="fps-
value">60</span>fps</span><br>
                <!-- <label for="motor1">Motor 1 <span id="dark">(<span
id="motor1-value">90</span>°)</span></label>
                <input type="range" id="motor1" min="0" max="180"
value="90"> -->
                <label for="gcode-input">Run G-Code Command:</label>
                <div id="slider-container", style="display: block;">
                    <input type="text" id="gcode-input" name="gcode-
input" value="G1 A90 B90 C90 D90 E90 F0">
                    <button id="run-gcode">SEND</button>
                </div>

                <div id="buttons-container">
                    <button id="start-robot">START</button>
                    <button id="pause-robot">PAUSE</button>
                    <button id="stop-robot">STOP</button>
                    <!-- sync with robot cehckbox -->
                    <input type="checkbox" id="sync-robot" name="sync-
robot" value="sync-robot">
                    <label for="sync-robot">Sync with robot</label>
                    <div id="heart-beat"></div>
                </div>
                <br>
                <label for="eepos-value">End effector position:  
<span id="dark"><span id="eepos-value">0</span></span></label>
            </div>
        </div>
    </main>
    <script src="sketch.js"></script>
</body>
</html>
```

JavaScript

```
let runGCodeButton;
let gcodeInput;
let heartBeatDiv;
let heartBeatTime = 0;
let syncCheckbox;

let motors = {
    m0: 90,
    m1: 90,
    m2: 90,
    m3: 90,
    m4: 90
};

let motorsGoal = {
    m0: 90,
    m1: 90,
    m2: 90,
    m3: 90,
    m4: 90
};

let gripperClosed = true;
let myFont;
let timeToUpdate = 0;
let togglePos = false;

//camera controls
let cam;
let isPanning = false;
let lastMouseX, lastMouseY;
let camAngle = 90;
let camHeight = 150;
let zoomLevel = 200;
let zoomGoal = 200;

endEffectorPos = {
    x: 0,
    y: 0,
    z: 0
};

//let server = "https://robot.xpiti.com";
//let server = "http://192.168.1.193";
let server = "";

let filesToLoad = [
    { type: 'model', key: 'armModelSeg0', path: 'arm/0.stl' },
```

```
    { type: 'model', key: 'armModelSeg1', path: 'arm/1.stl' },
    { type: 'model', key: 'armModelSeg2', path: 'arm/23.stl' },
    { type: 'model', key: 'armModelSeg3', path: 'arm/23.stl' }, // armModelSeg3 reuses
armModelSeg2
    { type: 'model', key: 'armModelSeg4', path: 'arm/4.stl' },
    { type: 'model', key: 'armModelSeg4_open', path: 'arm/4_open.stl' },
    { type: 'image', key: 'armTexture', path: 'arm/arm_texture.png' }
    // { type: 'font', key: 'myFont', path: 'Roboto-Regular.ttf' },
];

let assets = {}; // To store loaded assets
let retryLimit = 3;
let filesLoaded = false;

function loadFileSequentially(index = 0, retries = 0) {
    if (index >= filesToLoad.length) {
        console.log("All files loaded successfully!");
        filesLoaded = true;
        // start syncing with the robot (enable the checkbox)
        syncCheckbox.checked(true);
        return;
    }

    let file = filesToLoad[index];
    console.log(`Loading ${file.type} from ${file.path}...`);

    // Choose the correct loading function based on type
    let loadFunction;
    switch (file.type) {
        case 'model':
        loadFunction = loadModel;
        break;
        case 'image':
        loadFunction = loadImage;
        break;
        case 'font':
        loadFunction = loadFont;
        break;
        default:
        console.error(`Unknown type: ${file.type}`);
        return;
    }

    // Load the file
    loadFunction(file.path,
        (loadedAsset) => {
            // Success
            assets[file.key] = loadedAsset;
            console.log(`${file.key} loaded successfully.`);
            // Handle cases like armModelSeg3 depending on another
```

```
            if (file.key === 'armModelSeg3') {
                assets[file.key] = assets['armModelSeg2'];
            }
            loadFileSequentially(index + 1);
        },
        (err) => {
            // Failure
            if (retries < retryLimit) {
                console.warn(`Retrying to load ${file.path}... Attempt ${retries + 1}`);
                loadFileSequentially(index, retries + 1);
            } else {
                console.error(`Failed to load ${file.path} after ${retryLimit} attempts.`);
                loadFileSequentially(index + 1); // Skip to the next file
            }
        }
    );
}


function setup() {
    // Select the container element
    const container = document.getElementById("canvas-container");

    // Get the container's width and height
    const width = container.offsetWidth;
    const height = container.offsetHeight;

    // Create the canvas with the container's dimensions
    canvas = createCanvas(width, height, WEBGL);

    // Place the canvas in the container
    canvas.parent("canvas-container");

    cam = createCamera();
    setCamera(cam);

    //-----------------------------------------------
    // Link the slider from the HTML to the JavaScript
    runGCodeButton = select('#run-gcode');
    runGCodeButton.mousePressed(processGCodeInput);
    gcodeInput = select('#gcode-input');
    gcodeInput.changed(processGCodeInput);

    //start pause and stop buttons
    let startButton = select('#start-robot');
    startButton.mousePressed(() => {
        sendGCodeCommand("START");
    });
    let pauseButton = select('#pause-robot');
    pauseButton.mousePressed(() => {
```

```
        sendGCodeCommand("PAUSE");
    });
    let stopButton = select('#stop-robot');
    stopButton.mousePressed(() => {
        sendGCodeCommand("STOP");
    });

    syncCheckbox = select('#sync-robot');
    heartBeatDiv = select('#heart-beat');

    // armModelSeg0 = loadModel('arm/0.stl');
    // armModelSeg1 = loadModel('arm/1.stl');
    // armModelSeg2 = loadModel('arm/23.stl');
    // armModelSeg3 = armModelSeg2;
    // armModelSeg4 = loadModel('arm/4.stl');
    // armModelSeg4_open = loadModel('arm/4_open.stl');
    // armTexture = loadImage('arm/arm_texture.png');
    // myFont = loadFont('Roboto-Regular.ttf');

    loadFileSequentially();

    //textFont(myFont);

    angleMode(DEGREES);

    print("Setup complete");
}

function windowResized() {
    // Recalculate the dimensions when the window is resized
    const container = document.getElementById("canvas-container");
    const width = container.offsetWidth;
    const height = container.offsetHeight;

    // Resize the canvas to fit the container
    resizeCanvas(width, height);
}

function processGCodeInput(){
    let gcode = document.getElementById("gcode-input").value;
    sendGCodeCommand(gcode);
}


// Function to fetch motor positions (non-blocking)
function fetchMotorPositions() {
    // Make a GET request to the server
    fetch(`${server}/api/servo`)
        .then(response => {
            // Check if the response is ok
```

```
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }
            return response.json(); // Parse JSON response
        })
        .then(motorData => {
            // Update the motor variables
            // check if the motorData is not empty
            if(motorData.servo0 == undefined){
                return;
            }
            motorsGoal.m0 = motorData.servo0;
            motorsGoal.m1 = motorData.servo1;
            motorsGoal.m2 = motorData.servo2;
            motorsGoal.m3 = motorData.servo3;
            motorsGoal.m4 = motorData.servo4;
            gripperClosed = motorData.servo4 > 85;

            heartBeatTime = millis();

            //console.log('Motor positions updated:', motors);
        })
        .catch(error => {
            console.error('Error fetching motor positions:', error);
        });
}

// Function to send a G-Code command (non-blocking) with retries
function sendGCodeCommand(cmd) {
    // Define the endpoint
    const endpoint = `${server}/api/gcode`;

    const payload = {
        command: cmd, // The G-Code command string
    };

    console.log("Sending G-Code command:", JSON.stringify(payload));

    // Define the maximum number of retries
    const maxRetries = 3;

    // Function to attempt the request
    function attemptRequest(retriesLeft) {
        // Make a POST request to the server
        fetch(endpoint, {
            method: "POST", // Use the POST HTTP method
            headers: {
                "Content-Type": "application/json", // Specify JSON content type
            },
            body: JSON.stringify(payload), // Convert the payload to JSON
```

```
        })
            .then(response => {
                // Check if the response is ok
                if (!response.ok) {
                    throw new Error(`HTTP error! status: ${response.status}`);
                }
                return response.json(); // Parse JSON response
            })
            .then(data => {
                // Handle the server's response
                console.log("G-Code command sent successfully:", data);
                gcodeInput.style('background-color', '#1b1b1b');
            })
            .catch(error => {
                console.error(`Error sending G-Code command: ${error}`);

                // Retry if there are retries left
                if (retriesLeft > 0) {
                    console.log(`Retrying... Attempts left: ${retriesLeft}`);
                    attemptRequest(retriesLeft - 1); // Retry the request
                } else {
                    console.error("Max retries reached. G-Code command failed.");
                    gcodeInput.style('background-color', '#c21717');
                }
            });
    }

    // Start the first attempt with maxRetries
    attemptRequest(maxRetries);
}


function draw() {
    background(20);

    let hbt = millis() - (heartBeatTime + 500);
    hbt = constrain(hbt, 0, 1000);
    let heartBeatColor = map(hbt, 0, 2000, 200, 20);
    heartBeatDiv.style('background-color', `rgb(0, ${heartBeatColor}, 0)`);

    if(timeToUpdate < millis()){
        timeToUpdate = millis() + 1000;
        if(syncCheckbox.checked()){
            fetchMotorPositions();
        }
    }

    if(filesLoaded){
        //lerp motors for smooth visualisation
        motors.m0 = movePosition(motors.m0, motorsGoal.m0, 1);
```

```
        motors.m1 = movePosition(motors.m1, motorsGoal.m1, 1);
        motors.m2 = movePosition(motors.m2, motorsGoal.m2, 1);
        motors.m3 = movePosition(motors.m3, motorsGoal.m3, 1);

        endEffectorPos = getEndEffectorPosition();

        // Drawing
        push();
        lights();
        //translate(0, 100, 0); // Move the camera back
        // rotateX(-22);
        // rotateY(30*sin(frameCount/2));
        scale(1, -1, 1);
        drawAxes(this, 100);
        drawGrid(this, 500, 10);
        drawGrid(this, 1000, 50);
        drawScene(this, endEffectorPos);
        pop();

        document.getElementById("fps-value").textContent = round(frameRate());
        document.getElementById("eepos-value").textContent = `x: ${round(endEffectorPos.x)}mm,
y: ${round(endEffectorPos.y)}mm, z: ${round(endEffectorPos.z)}mm`;
    }else{
        noFill();
        stroke(255);
        strokeWeight(3);

        let rot = millis() / 2;
        let a1 = map(sin(millis()/5), -1, 1, 0, 180) + rot;
        let a2 = map(sin(millis()/6), -1, 1, 180, 360) + rot;
        arc(0, -75, 50, 50, a1, a2);
        noStroke();
        fill(50);
        box(200, 1, 100);
    }


    //lerp zoom
    zoomLevel = lerp(zoomLevel, zoomGoal, 0.05);
    // Handle rotation when no key is pressed
    if (mouseIsPressed && mouseX > 0 && mouseX < width && mouseY > 0 && mouseY < height) {
        let dx = mouseX - lastMouseX;
        let dy = mouseY - lastMouseY;
        camAngle += dx * 0.1;
        camHeight += dy * 0.5;
        camHeight = constrain(camHeight, 0, 500);
    }

    cam.camera(cos(camAngle)*zoomLevel, -camHeight, sin(camAngle)*zoomLevel, 0, -camHeight/2, 0,
0, 1, 0);
```

```
    // Update the last mouse position for rotation
    lastMouseX = mouseX;
    lastMouseY = mouseY;
}

function animateRoot(){
    motors.m0 = ( sin(frameCount/3.22) * 90 ) + 90;
    motors.m1 = ( (sin(frameCount/3.9112) * sin(frameCount/1.432)) * 90 ) + 90;
    motors.m2 = ( sin(frameCount/1.55) * 90 ) + 90;
    motors.m3 = ( sin(frameCount/2.32) * 90 ) + 90;
}

function drawGrid(buffer, size, spacing){
    let gridCount = size/spacing/2;

    buffer.stroke(255, 33);
    buffer.strokeWeight(1);
    for(let i = -gridCount; i <= gridCount; i++){
        buffer.line(-spacing * gridCount, 0, i * spacing, spacing * gridCount, 0, i * spacing);
        buffer.line(i * spacing, 0, -spacing * gridCount, i * spacing, 0, spacing * gridCount);
    }
}

function drawScene(buffer, eePos){
    // Draw the robot arm
    drawRobot(buffer, motors);


    // Draw the end effector position
    buffer.push();
    buffer.fill(255, 0, 0);
    buffer.translate(eePos.x, eePos.y, eePos.z);
    buffer.sphere(3);
    drawAxes(buffer, 20);
    buffer.pop();

    buffer.push();

    buffer.stroke(0);
    buffer.fill(100, 0, 150);
    buffer.translate(-72, 15, 100);
    buffer.box(30, 30, 50);

    buffer.fill(200, 0, 0);
    buffer.translate(33.5, 0, 0);
    buffer.box(30, 30, 50);

    buffer.fill(255, 150, 0);
    buffer.translate(33.5, 0, 0);
```

```
    buffer.box(30, 30, 50);

    buffer.fill(255, 255, 0);
    buffer.translate(33.5, 0, 0);
    buffer.box(30, 30, 50);

    buffer.fill(0, 255, 0);
    buffer.translate(33.5, 0, 0);
    buffer.box(30, 30, 50);

    buffer.pop();
}

function drawRobot(buffer, robotMotors) {
    let m0 = map(robotMotors.m0, 0, 180, -90, 90);
    let m1 = map(robotMotors.m1, 180, 0, -90, 90);
    let m2 = map(robotMotors.m2, 180, 0, -90, 90);
    let m3 = map(robotMotors.m3, 180, 0, -90, 90);


    let segmentHeight = 50;  // Height of each segment

    //buffer.texture(armTexture);
    buffer.fill(255);
    buffer.noStroke();

    // --- Arm Segment 0 (Base) ---
    buffer.push();

    buffer.model(assets["armModelSeg0"]); // Draw stationary base
    buffer.pop();

    // --- Arm Segment 1 ---
    buffer.push();
    buffer.translate(0, segmentHeight, 0);  // Move armModelSeg1 above the base by 46.25 mm
    buffer.rotateY(m0);               // Rotate armModelSeg1 horizontally around Y-axis (base
rotation)
    buffer.model(assets["armModelSeg1"]);

    // --- Arm Segment 2 ---
    buffer.rotateX(m1);                 // Rotate armModelSeg2 vertically around X-axis
    buffer.model(assets["armModelSeg2"]);

    // --- Arm Segment 3 ---
    buffer.translate(0, segmentHeight, 0); // Move armModelSeg3 above armModelSeg2 by 50mm
    buffer.rotateX(m2);                 // Rotate armModelSeg3 vertically around X-axis
    buffer.model(assets["armModelSeg3"]);

    // --- Arm Segment 4 (End Effector) ---
```

```
    buffer.translate(0, segmentHeight, 0); // Move armModelSeg4 (end effector) above
armModelSeg3 by 50mm
    buffer.rotateX(m3);                 // Rotate the end effector
    if(gripperClosed){
        buffer.model(assets["armModelSeg4"]);
    }else{
        buffer.model(assets["armModelSeg4_open"]);
    }

    buffer.fill(0);

    buffer.translate(0, segmentHeight, 0); // Move the end effector above armModelSeg4 by 50mm
    // drawAxes(10);
    // buffer.fill(200);
    // buffer.noStroke();
    // buffer.sphere(3);

    buffer.pop();  // End of robot arm drawing

    buffer.fill(0);
}

function drawAxes(buffer, axisLength) {
    buffer.strokeWeight(1.5);
    buffer.stroke(255, 0, 0);
    buffer.line(0, 0, 0, axisLength, 0, 0);
    buffer.stroke(0, 255, 0);
    buffer.line(0, 0, 0, 0, axisLength, 0);
    buffer.stroke(0, 0, 255);
    buffer.line(0, 0, 0, 0, 0, axisLength);

    buffer.noStroke();
    buffer.push();
    buffer.fill(255, 0, 0);
    buffer.translate(axisLength, 0, 0);
    buffer.rotateZ(-90);
    buffer.cone(3, 8);
    buffer.pop();

    buffer.push();
    buffer.fill(0, 255, 0);
    buffer.translate(0, axisLength, 0);
    buffer.cone(3, 8);
    buffer.pop();

    buffer.push();
    buffer.fill(0, 0, 255);
    buffer.translate(0, 0, axisLength);
    buffer.rotateX(90);
    buffer.cone(3, 8);
```

```
    buffer.pop();
}

function getEndEffectorPosition() {
    // base is at 0,0,0,
    let L1 = 50; //first segment, rotating horizontally
    let L2 = 50;
    let L3 = 50;
    let L4 = 75; //end effector

    let A1 = 0 - motors.m0;
    let A2 = motors.m1 - 180;
    let A3 = 90 + motors.m1 + motors.m2;
    let A4 = motors.m1 + motors.m2 + motors.m3;

    let d = cos(A2)*L2 + cos(A3)*L3 + cos(A4)*L4;
    let X = cos(A1) * d;
    let Y = L1 + sin(A2)*L2 + sin(A3)*L3 + sin(A4)*L4;
    let Z = sin(A1) * d;

    return {
        x: X,
        y: -Y + 100,
        z: Z
    }
}

function mouseWheel(event) {
    // Zoom in/out when scrolling
    zoomGoal += event.delta * 0.5;  // Adjust zoom speed here
    zoomGoal = constrain(zoomGoal, 50, 500); // Constrain zoom to desired limits
}

function movePosition(pos, goal, speed){
    if(abs(pos - goal) <= speed){
        return goal;
    }

    if(pos < goal){
        return pos + speed;
    }else{
        return pos - speed;
    }
}
```