Two-axis Robotic Arm

Kellen A Newby **934-536-148**
Dario Mejia Garduno **934-522-234**
Cory M Towner **934-551-224**

Got feedback on this document? Email us!

Top-Level Architecture Block Diagram

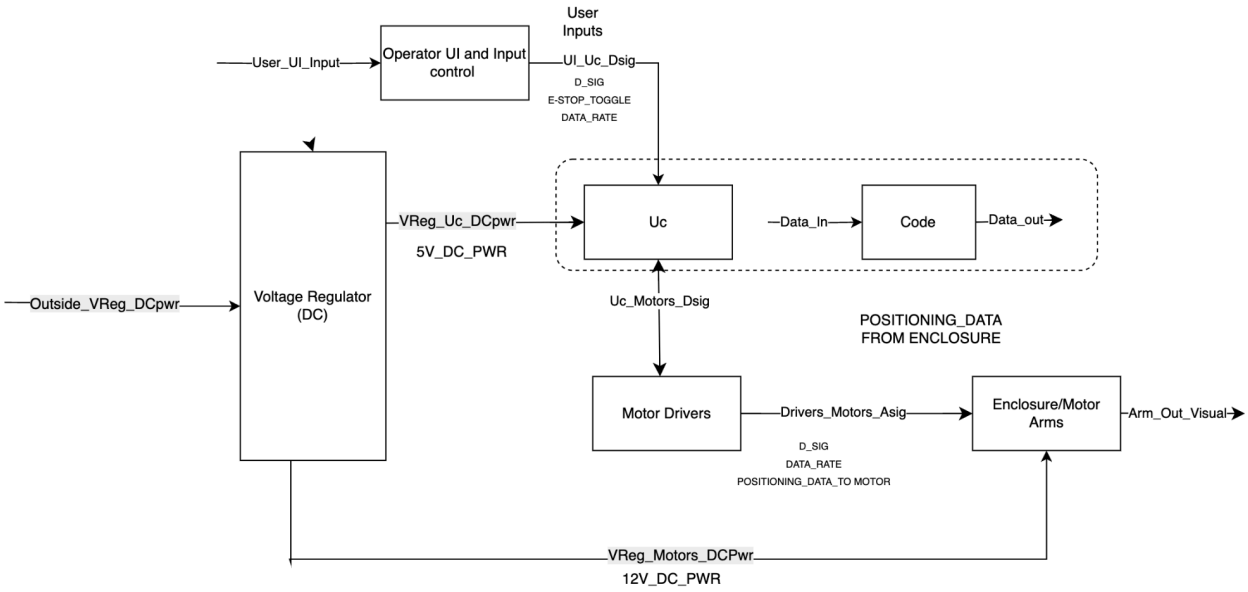

Figure 1: Two-Axis Robot Arm Top-Level Diagram

Code Block

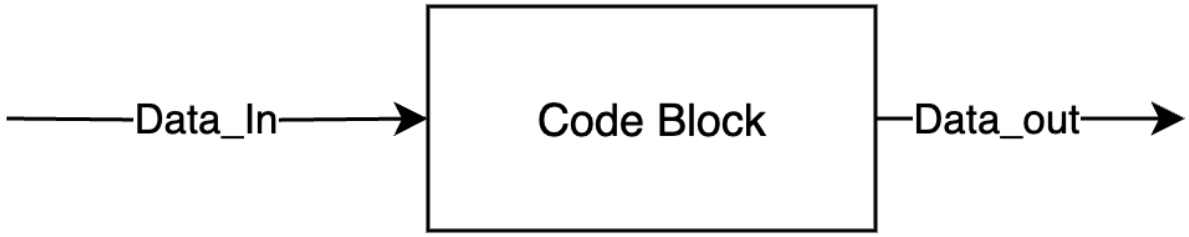

Figure 2: Code Block Diagram

Video Link

WIP

Description

The code block of the Two Axis Robot Arm is the core processing unit responsible for interpreting and parsing G-Code into stepper motor movements (mapping trajectory of the robot arm using Bresenham's Algorithm [4]). Additionally the code block also interfaces with a Web Server, managing file uploads/downloads for preset drawings that can be saved in the Web Server. The Code Block also provides a stand-by routine, polling for button inputs to determine the target routine to be processed. Lastly, the Code Block implements swift halt of any current program when the E-Stop is pressed through robust Interr.

Design Details

The Code Block for the Two-Axis Robot Arm is implemented in C++ using the Arduino IDE within the PlatformIO environment. This block is responsible for processing user and web-based inputs [6], interpret G-Code files, compute synchronized trajectories for the robot arm's two axes, and ensure safe, real-time operation through robust interrupt handling combined with SPIFFS journaling [5] to guarantee system integrity.

Core Functionality

Upon startup, the code executes an initialization routine responsible for configuring all relevant GPIO pins, establishes a WiFi connection and launches the web server, ensuring the system is network-accessible before proceeding. Lastly, the INIT routine also mounts the SPIFFS file system to enable quick access to uploaded G-Code files.
The main operational flow is divided into several stages:

Idle/User Input Stage:

The Idle/User Input stage is executed as the main loop of the program after initialization. In this stage the system continuously polls the user buttons to determine the next action. If a drawing routine is selected—either from preset options or a newly uploaded G-Code file—the system transitions to the drawing stage.

Configuration Routine:

Via the web interface, users can upload new G-Code files or adjust preset drawing parameters. The web server endpoints handle these requests, storing files in SPIFFS [5] and updating configuration variables in real time.

Drawing Execution Stage:

Once a drawing is initiated, the code parses the G-Code line by line into coordinated stepper motor movements. Bresenham's algorithm [4] is implemented to interpolate between coordinate points, ensuring smooth, synchronized motion of both axes. The current position of the arm is tracked in software, allowing for recovery if an E-Stop is triggered. Once a drawing routine is finalized the program returns to the main idle ready to accept new user or web-based commands.

Emergency Stop Handling:

The Emergency Stop Routine operates as an ISR (Interrupt Service Routine) which executes a two-level stop, disabling the motor drivers (hardware cutoff), and halting all motion routines (software halt). The portENTER_CRITICAL_ISR() blocks lower-priority interrupts from occurring during the E-Stop operation. The ISR uses portENTER_CRITICAL_ISR() to block lower-priority interrupts and esp_task_wdt_reset() to safely interact with the watchdog timer before forcing a system reset with esp_restart(). This two-level stop guarantees both immediate physical safety and software recovery.

Inputs

User Buttons (Start, Preset Selection):
Physical buttons connected to ESP32 GPIO pins are polled in the main loop. These buttons allow the user to initiate drawing routines or select preset drawings.

Web Server (G-Code File Upload):
The ESP32 hosts a local web server using the <WiFi.h> and <WebServer.h> libraries, accessible via WiFi. Users connect through a browser to upload G-Code files or configure drawing parameters, with files stored in SPIFFS for persistent access.

E-Stop Button:
A dedicated emergency stop is connected to a GPIO pin (e.g., GPIO 0) with a pull-up resistor. This input is monitored by a hardware interrupt for immediate response.

Interface Validation

Interface property ***Be sure to use the naming convention "from_to_type" here.**	**Why is this interface property this value?**	**How do you know your design details will meet or exceed this property? Cite your sources in IEEE.**

Code_Data-In

E-Stop Interrupt	The Emergency Stop is executed as an ISR that performs a two-level stop. This implementation ensures immediate response pausing all movement routines	The ISR resides in IRAM for minimal latency, ensuring the fastest possible response time . ISR guarantees the Emergency Stop will be executed regardless of the current state of the program.
G-Code	G-code is a widely used language to	Each G-Code command is translated into step and direction signals for the stepper motors, using coordinate interpolation for smooth trajectories. Bresenham's Algorithm [4]is used for calculating trajectories between points.
Preset Drawings	A selection of stared drawing routines are made available to the user	Users may select from saved drawing routines. This routines are stored in non-volatile memory SPIFFS
Web Server	The <WiFi.h> and <WebServer.h> libraries facilitate interfacing with a locally hosted web server. The server provides a readily accessible method for file management and user interaction.	The ESP32's built-in web server libraries (<WiFi.h>, <WebServer.h>) enable asynchronous HTTP handling [6], ensuring reliable data transfer. The server is tested to handle concurrent connections and file uploads without blocking the main control loop.

Code_Data-Out

Preset Drawings	The system must be able to store at least three preset drawings the User can choose from, or even configure through the web server.	Preset drawings will be stored using SPIFFS [5], providing persistent non-volatile storage directly in the ESP32's flash, additionally through the web server, user may also configure the preset drawings to different files of their choice

Movement Calculation	Parsing the G-Code is necessary to determine the movements necessary to execute a drawing task.	Bresenham Algorithm [4] ensures proper interpolation for the motion of the Robotic Arm
Software Halt	The Software must halt any current routine after the E-Stop is activated.	The ISR implementation guarantees an immediate response pausing all movement routines.

Verification Process

1. Access PlatformIO, and open the ECE342/TwoAxisRoboticArm project..
    a. We will be using test_main.cpp to test the Estop and the GCode processor
2. Build the project with the CTRL-ALT-B command
3. Flash the ESP32 with the command CTRL-ALT-U
4. Open serial monitor with CTRL-ALT-U
    a. At the stage the program is already running with hardcoded G Code Commands (Preset Drawings)
5. Press e to trigger the E-stop
6. Verify an "[E-STOP] triggered!" message is displayed on the Serial Monitor.
7. Now press "r" to resume previous operation, or "o" to move the robotic arm to origin position.
8. **If you press r** , verify [Resuming] Continuing to Xn Yn.. message is printed to the monitor
    a. Where n is just a placeholder for a whole number coordinate.
9. Verify that the nest printed line shows the coordinates your are moving to, and the feedrate
10. Lately Verify the bottom line shows a time-stamp follow by the GCode command with the x and y coordinate, feed rate and mode display to the terminal
    a. Ex. 435257,G1 Command,30.00,0.00,100.00,ABS
11. **If you pressed "o" verify** a message formatted as follows is printed to the terminal
    a. [Returning to Origin]...
    b. → Moving to X=0.00, Y=0.00 @ Feed Rate=100.00
    c. 643503,G1 Command,0.00,0.00,100.00,ABS
    d. [System] State reset. Position = (0,0), Mode = ABS
    e. 643804,Return to Origin,0.00,0.00,100.00,ABS

12. CTRL - C to exit Serial Monitor
13. Compare the printed motor commands to the expected values based on the G-Code and Bresenham's algorithm.
14. Test different G-Code commands and verify the correct number of steps, direction, and timing are calculated.

Artifacts

1. ESP32 Web Server Implementation
Researched how to set up and serve a web interface on the ESP32 using the Arduino IDE and <WiFi.h> and <WebServer.h>.

Read through step-by-step tutorials to understand HTTP request handling, client connections, and how to output diagnostic information to the Serial Monitor for debugging.

Example code from Random Nerd Tutorials demonstrated how to control GPIO pins via web requests, which was adapted for file upload and G-Code management for this project.
Researched SPIFFS (Serial Peripheral Interface Flash File System) on ESP32
Studied how to initialize, mount, and use SPIFFS for persistent file storage on the ESP32.

3. G-Code Parsing and Interpretation
Reviewed the structure and syntax of G-Code commands (e.g., G0, G1, X/Y/Z coordinates, F for feedrate).
Consulted open-source robot arm controller projects and documentation
Explored Python-based G-Code generation and GUI tools for creating and testing drawing routines.

4. Bresenham's Line Algorithm for Trajectory Planning

Learned about Bresenham's algorithm for efficient, integer-based line interpolation between two points, ensuring smooth and synchronized axis movement.
Used pseudocode and online tutorials to reinforce my understanding of the new tools I am not very familiar with, primarily G-Code and Motor Driver interfacing.

5. Reference Schematics and Block Diagrams

Consulted reference schematics for ESP32 DevKit V1 pinouts, L298N motor driver wiring, and NEMA 17 stepper motor connections.
Iterated on block diagrams to clarify the data flow between the Code Block, Microcontroller, Motor Drivers, Web Server, and User Inputs.

6. Prior Coursework

Reviewed lab assignments (Project Document – ECE 341) for documentation standards and block interface conventions. As well as strategies that worked well in the past I can carry on to this project, and have a smooth start.

Future Recommendations

I think this far, I have become more comfortable with the overall structure of these block documents, understanding the layout and the process of populating them as tools I can use to track progress in our project, and maintain a well laid out overview of what we have accomplished, what the goals are, and the tools we have.

Admittedly, I underestimated how much time I would spend just researching. Although, I feel I have a very good roadmap for implementing the Code Block, I think I would've likely benefited more had I started coding earlier rather than spending too much time reading and researching all at once. I would tell myself at the beginning of the term to break down my goals with the code block as soon as possible and tackle them one at a time, to make sure I can progress as I learn and research more.

References

[1] Espressif Systems, "ESP32 Series Datasheet," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf [Accessed: Apr. 20, 2025]

[2] Espressif Systems, "ESP-WROOM-32 Datasheet." https://www.mouser.com/datasheet/2/891/esp-wroom-32_datasheet_en-1223836.pdf?srsltid=AfmBOoq51T8_Wn243PBMdjcEAsQfkJYYEVY2hw_7I4RFvrVibQ8lw0lm](https://www.mouser.com/datasheet/2/891/esp-wroom-32_datasheet_en-1223836.pdf?srsltid=AfmBOoq51T8_Wn243PBMdjcEAsQfkJYYEVY2hw_7I4RFvrVibQ8lw0lm)

[3] Arduino Kit Project, "Learn DC Motor Control with ESP32 and L298N Driver," Arduino Kit Project, [Online]. Available: https://arduinokitproject.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/ [Accessed: Apr. 20, 2025].

[4] GeeksforGeeks, "Bresenham's Line Generation Algorithm," GeeksforGeeks, Mar. 11, 2024. [Online]. Available: https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/ [Accessed: Apr. 20, 2025]

[5] Espressif Systems, "SPIFFS Filesystem — ESP-IDF Programming Guide v5.2.1 documentation," Espressif, [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/spiffs.html [Accessed: Apr. 20, 2025].

[6] R. Santos, "ESP32 Web Server – Arduino IDE," Random Nerd Tutorials, [Online]. Available: https://randomnerdtutorials.com/esp32-web-server-arduino-ide/ [Accessed: Apr. 20, 2025].

[7] Arduino Kit Project, "Learn DC Motor Control with ESP32 and L298N Driver," Arduino Kit Project, [Online]. Available: https://arduinokitproject.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/ [Accessed: Apr. 20, 2025].

[8] R. Santos and S. Santos, "ESP32 Servo Motor Web Server with Arduino IDE," Random Nerd Tutorials, Jun. 13, 2024. [Online]. Available: https://randomnerdtutorials.com/esp32-servo-motor-web-server-arduino-ide/ [Accessed: Apr. 20, 2025].

[9] Espressif Systems, "HTTP Server — ESP-IDF Programming Guide documentation," Espressif, [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/protocols/esp_http_server.html [Accessed: Apr. 20, 2025].

[10] PicoBricks, "Two Axis Robot Arm Project with PicoBricks," PicoBricks, [Online]. Available: https://picobricks.com/blogs/robotic-stem-projects/two-axis-robot-arm/ [Accessed: Apr. 20, 2025].

Microcontroller Block

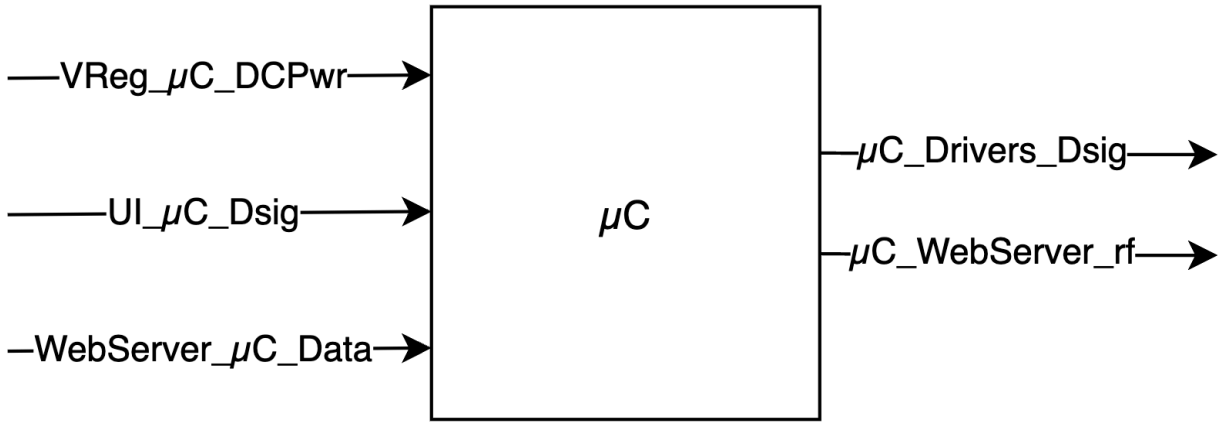

Figure 3: Microcontroller Block Diagram

Video Link

WIP

Description

The Microcontroller (µC) Block is the central processing unit of the Two-Axis Robotic Arm. It governs the system's behavior, receiving commands, processing input signals, generating motor control signals, and ensuring safe and accurate motion. This block uses the ESP32-WROOM-32 [12] microcontroller and communicates with nearly every subsystem, making it essential to system integration and control.

The microcontroller is responsible for establishing a connection with the Web Server that enables receiving G-code commands, and files via WiFi [12, p. 3]. Additionally it also receives information from

The microcontroller inputs are a the following

- **WebServer_µC_rf (input)**: Receives G-code commands, configuration files, and drawing instructions via WiFi from a local browser-based GUI.
- **UI_µC_dsig (input)**: Reads digital button inputs for commands such as drawing initiation, routine selection, and emergency stop.
- **VReg_µC_DCpwr (input)**: Receives regulated 5V DC power that is stepped down to 3.3V by the onboard voltage regulator for proper operation

The outputs are the following:

- **µC_Drivers_dsig (output)**: Sends digital step and direction signals to the stepper motor drivers (L298N) to control the SCARA robot arms.

- **µC_WebServer_rf (output)**: Provides status and responses back to the server (e.g., file upload, drawing progress [state]).

Through these interfaces, the µC interprets incoming G-code files, computes interpolated joint movements, coordinates real-time motor actuation, and ensures user safety with a prioritized emergency stop routine. It also manages persistent file storage for preset drawings using the SPIFFS (Serial Peripheral Interface Flash File System [5]) available on the ESP32

Design Details

The ESP32-WROOM-32 microcontroller functions as the system's control hub. It is programmed using C++ in the PlatformIO environment and utilizes the Arduino framework for compatibility with embedded libraries such as <WiFi.h>, <WebServer.h>, and SPIFFS. The µC performs the following functional tasks:

1. User Interface Input
    - Buttons (e.g., Start, Preset Select, E-Stop) are connected to ESP32 GPIOs with pull-up resistors and are read via digital polling or interrupt service routines.
    - The E-Stop specifically triggers a critical ISR using portENTER_CRITICAL_ISR() to prevent other interrupts from interfering. This halts motor output and resets the system via esp_restart() [11][12].

2. Wireless Communication via Web Server
    - The µC sets up a local web server that listens for HTTP requests over WiFi. This server uses WebServer.h and asynchronous callbacks to manage:
    - File uploads (G-code files)
    - Parameter configuration
    - Control commands (start, stop, resume) [6][9].
    - Files uploaded are stored on SPIFFS and accessed line-by-line during execution.

3. Parsing and Interpreting G-code
    - The microcontroller reads each G-code line (G0, G1, etc.), parses its parameters (X, Y, F), and translates it into physical movement using Bresenham's algorithm [4].
    - G90 and G91 are interpreted to switch between absolute and relative positioning modes.
    - G20/G21 control unit interpretation (inches/mm).

4. Motion Execution

- The parsed movements are synchronized and converted into step and direction pulses for the L298N stepper drivers, controlling Nema17 motors.
- Timing of these pulses is calculated from feedrate F, and physical constraints (gear ratio, microstepping, etc.) are used to generate proper pulse width and delay [3][14].

5. Preset Drawings and Non-Volatile Storage

- At least three preset drawings are stored persistently on SPIFFS [5] and can be selected through buttons or the web interface.
- Users may override preset files through the GUI, ensuring flexibility while meeting the preset drawing requirement.

6. Power Handling

- The ESP32 is powered from a 3.3V regulated supply, typically derived from a linear regulator (NCP111[13]). Current consumption is nominally 80 mA [12, p. 2] and peaks at 160 mA during WiFi or highly demanding operations [12].

Alternative Designs

While the current design uses WiFi for G-code delivery and SPIFFS for storage, alternative designs could include:

- UART Serial interface for a PC-based GUI when wireless access is limited.
- SD Card storage as an alternative to SPIFFS for greater file size flexibility and standard file management tools.
- Dedicated RTOS task scheduling using FreeRTOS to better prioritize safety-critical operations (like E-Stop) and file management concurrently [11].

Interface Validation

Interface property *Be sure to use the naming convention “from_to_type” here.	Why is this interface property this value?	How do you know your design details will meet or exceed this property? Cite your sources in IEEE.

VReg_µC_DCpwr: Input

V_{min}: 4.75V	Allowable dropout tolerance of onboard LDO (NCP1117 [13]) on ESP32	Datasheet [13, p. 3] specifies 1.2V max dropout for 5V output; simulation confirms LDO stability at 4.75V [11].
V_{MAX}: 5.25	Accounts for 5% voltage tolerance to protect µC. Expected input from the Voltage Regulator block is 5V.	The onboard regulator is part of th the NCP1117 series which can operate with up to 20 V input [13, p. 1]
$I_{Nominal}$: 50-80 mA	This interface is intended to power the ESP32 microcontroller, which has an average operational input current of 80 mA [12, p. 2]. Under low operation the typical current ma	Expected output voltage by the onboard Linear Regulator is 3.3V by the Datasheet [13, p. 3].

UI_µC_dsig: Input

Logic Level = 3.3V	ESP32 GPIO pins operate at 3.3V logic level.	ESP32 Datasheet [11, p. 52] specifies
Active Level = LOW	Pull- up resistors are used, meaning that button press pulls to ground.	ESP32 uses internal pull-ups [2]
Emergency Stop ISR Falling Edge Detection	Ensures immediate system halt upon user-initiated E-Stop signal	Implemented using ESP32 GPIO interrupt; tested by triggering the E-Stop and observing system response time; confirmed via Serial Monitor logs and system behavior

WebServer_µC_rf: Input

HTTP 1.1 Protocol	Standard protocol for local ESP32 WebServer	ESP-IDF and Arduino core support HTTP/1.1 through <WebServer.h> [6], [9].
SPIFFS	SPIFFS (Serial Peripheral Interface Flash File System) is memory management on the ESP32, superseding the deprecated EEPROM [16]. SPIFFS is used for storing files shared across the Web Server and the Microcontroller.	SPIFFS is well-suited for embedded systems, IoT devices, and applications where flash memory is the primary storage medium [16]. ESP32 supports SPIFFS[5].
Data rate 150-800 Kbps	Although the ESP32 Series Datasheet mentions a theoretical data rate maximum of 150 Mbps [11, p. 33], therefore the selected range is reasonably appropriate for proper system operation.	The ESP32 microcontroller series are rated for up to 150 Mbps under 802.11n [11, p. 33].

µC_Drivers_dsig: Output

Logic Level = 3.3 V	ESP32 GPIOs operate at 3.3V logic. L298N accepts this input level to drive the H-bridge inputs.	The ESP-WROOM-32 operates GPIOs at 3.3V [12, p. 14]; L298N inputs follow standard TTL logic levels [17, p. 1].
Bit-Banged Step Control	Enables precise control of motor steps with software-generated pulses	GPIO pins can be sequenced to output directly to the IN1–IN4 pins to energize the motor coils in the correct order allowing precise motor operation.
Max Pulse Frequency ≈ 1 kHz	The Nema17 Stepper Motors Datasheet specifies 200 steps per	Nema 17 Motors are rated for 200 steps per revolution and rated for 3000 RPM [19]. Therefore staying at

	revolution [18, p. 1], with some sources specifying a maximum capability of 3000 RPM [19]. Nonetheless, the L298N is an older driver creating limitations when operating Stepper Motors like the Nema 17, therefore for reliable operation 300 RPM is sufficient. $\frac{300\ rev}{60\ sec} \times 200\frac{steps}{rev} = 1000\ \frac{steps}{sec}$	lower target of 300 RPM for proper operation of the motors during drawing routines sets the pulse frequency at 1KHz

µC_WebServer_rf: Output

HTTP 1.1 Protocol	Standard protocol for local ESP32 WebServer	ESP-IDF and Arduino core support HTTP/1.1 through <WebServer.h> [6], [9].
SPIFFS	SPIFFS (Serial Peripheral Interface Flash File System) is memory management on the ESP32, superseding the deprecated EEPROM [16]. SPIFFS is used for storing files shared across the Web Server and the Microcontroller.	SPIFFS is well-suited for embedded systems, IoT devices, and applications where flash memory is the primary storage medium [16]. ESP32 supports SPIFFS[5].
Data rate 30-500 Kbps	Although the ESP32 Series Datasheet mentions a theoretical data rate maximum of 150 Mbps [11, p. 33], therefore the selected range is reasonably appropriate for proper system operation.	The ESP32 microcontroller series are rated for up to 150 Mbps under 802.11n [11, p. 33].

Verification Process

VReg_µC_DCpwr: Input

Setup:

1. Power ESP32 through USB or by connecting it to a power supply through the 5V pin.
2. Flash the ESP32 with the following diagnostic code VReg_µC_DCpwr: Input Verification Code Disconnect USB power after uploading.

3. Connect 5V and GND from the power supply directly to the ESP32's 5V and GND pins.

5.00 V:

4. Set the bench power supply to 5.00 V
5. Power the ESP32.
6. Confirm:
    a. Check onboard LED blinks.
7. Let the test run for 30 seconds.

4.75 V :

8. Set the bench power supply to 5.00 V..
9. Repeat steps 6-8

5.25 V:

10. Set the bench power supply to 5.25 V.
11. Repeat steps 6-7
12. Let the test run for 30 seconds.

UI_µC_dsig: Input:

Setup:

1. Connect ESP32 board to laptop through USB.
2. Flash the following diagnostic code UI_µC_Dsig: Input Verification Code
3. Connect Power Supply ground to GND pin in microcontroller and prepare to connect to one of the jumper wires in GPIOs 14 or 27 to the power supply
4. Connect supply at 3.3 V to GPIO 14 (BUTTON1_PIN) and GND
    a. Verify BUTTON1_PIN is now: HIGH is printed to Serial Monitor
    b. Turn off Supply
    c. Verify BUTTON1_PIN is now: LOW is printed to Serial Monitor
5. Connect supply at 3.3 GPIO 27 (ESTOP_PIN) and GND
    a. Verify E-STOP] ISR Activated is printed to Serial Monitor
    b. Turn off Supply
    c. Verify ESTOP_PIN is now: LOW is printed to Serial Monitor

y

WebServer_µC_rf: Input/Output:

Setup:

1. Use the WebServer_µC_rf: Input/Output Verification Code snippet to:
    a. Mount SPIFFS.
    b. Connect to Wi-Fi.
    c. Set up a simple web server to handle file uploads.
    d. Log transfer durations to the Serial Monitor.

2. Use PlatformIO to upload the code to your ESP32 board.

3. Launch the Serial Monitor to observe logs.

SPIFFS:

4. Observe the Serial Monitor upon ESP32 startup.

5. Look for message: "SPIFFS Mounted Successful"

Wi-Fi Connection Verification:

6. After SPIFFS mount, the ESP32 should attempt to connect to Wi-Fi
7. Observe the Serial Monitor for the assigned IP address.

Web Server File Upload:

8. On your laptop, open a web browser and navigate to the ESP32's IP address.

9. Use the provided form to select and upload a test file (e.g., a small .txt file).

10. Check Serial Monitor for upload progress and completion messages.
    a. "Upload Start: filename"
    b. "Upload Progress: X bytes"
    c. "Upload End: Y bytes"

Data Transfer Rate:

11. Note the timestamps of upload start and end messages in the Serial Monitor.

12. Calculate the duration and divide the total bytes by the duration to get the transfer rate in bytes per second.

13. Assert the transfer rate falls within the expected range for your Wi-Fi network and file size.

µC_Drivers_dsig: Output

Setup**:**

1. Use the following µC_Drivers_dsig: Output Verification Code snippet to generate step pulses on a designated GPIO pin and log the pulse count to the Serial Monitor.
2. Adjust STEP_PIN to match the GPIO pin you intend to test.
3. Use PlatformIO to upload the code to your ESP32 board.
4. Launch the Serial Monitor to observe logs
5. Attach the oscilloscope probe to the designated GPIO pin (STEP_PIN).

6. Set the oscilloscope to capture and measure pulse characteristics.

   Pulse Frequency:
7. Observe the waveform on the oscilloscope.

8. Measure the time interval between consecutive rising edges (or falling edges) of the pulses.

    a. The measured interval should be approximately 1 millisecond, corresponding to a 1 kHz frequency.

   Pulse Width:

9. Use the oscilloscope to measure the duration of the HIGH state of each pulse.

10. Confirm that each pulse has a width of at least 10 µs.

11. Monitor the Serial output for pulse count increments.

12. Ensure the count increases consistently over time.

13. Confirm the Serial Monitor displays an incrementing pulse count, indicating successful pulse generation.

Artifacts

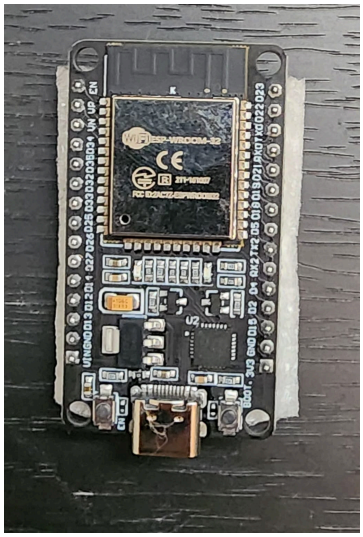

Figure 4: ESP32-Wroom-32 Dev Kit V1 Type C

Initially I was working with the Dev Kit V1 Type C board however the limited GPIO pin compared to other boards led me to decide to switch to the thinner Dev Kit C board.

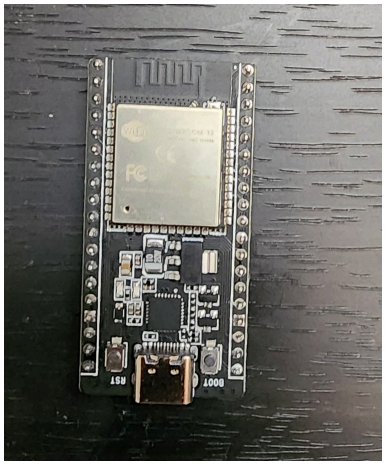

Figure 5: ESP32-Wroom-32 Dev Kit C

Switching to this board also allowed me to use a breakout board we already had available that fit the thinner dimensions of the Dev Kit C board.

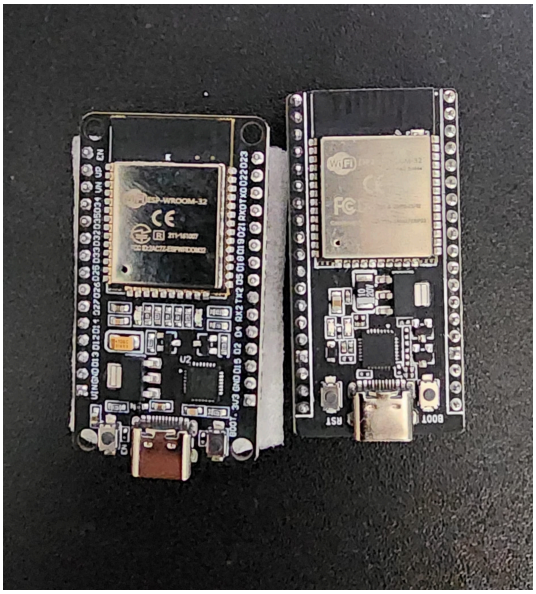

Figure 6: ESP32-Wroom-32 Boards Side by Side Comparison

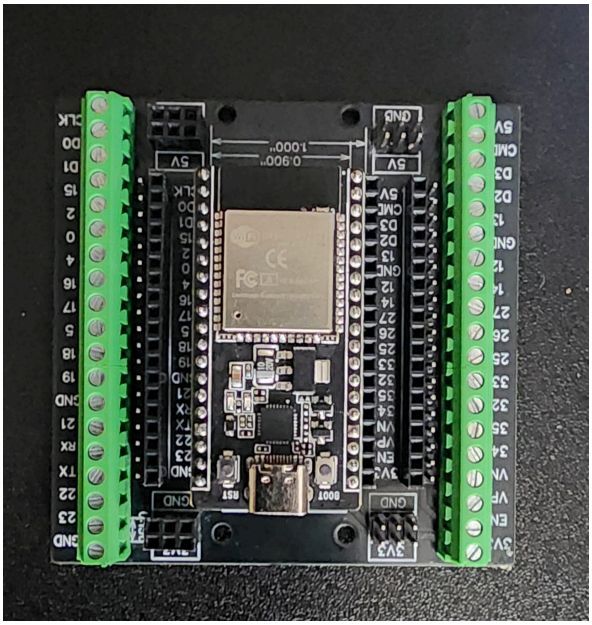

Figure 7: Breakout Board

The above breakout board facilitates testing as I can simply use jumper cables to easily connect to any of the boards pins.

VReg_µC_DCpwr: Input Verification Code

```
#include <Arduino.h>

void setup() {
  Serial.begin(115200);
  pinMode(2, OUTPUT); // Onboard LED for visual confirmation
}

void loop() {
  Serial.println("µC ACTIVE");
  digitalWrite(2, HIGH);
  delay(500);
  digitalWrite(2, LOW);
  delay(500);
}
```

UI_µC_Dsig: Input Verification Code

```
#ifdef TEST_UI_INPUTS

#include <Arduino.h>

#define BUTTON1_PIN 14
#define ESTOP_PIN 27
#define LED_PIN 2

bool prev_button_state = HIGH;
bool prev_estop_state = HIGH;
volatile bool estop_triggered = false;

void IRAM_ATTR handleEstop() {
  estop_triggered = true;
}

void setup() {
  Serial.begin(115200);
  pinMode(BUTTON1_PIN, INPUT);
  pinMode(ESTOP_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(ESTOP_PIN), handleEstop, FALLING);
  Serial.println("UI_µC_dsig Input Monitoring Initialized");
```

```
}

void loop() {
  // === Live Logic Level Reading ===
  int button_state = digitalRead(BUTTON1_PIN);
  int estop_state = digitalRead(ESTOP_PIN);

  if (button_state != prev_button_state) {
    Serial.print("BUTTON1_PIN is now: ");
    Serial.println(button_state == HIGH ? "HIGH" : "LOW");
    prev_button_state = button_state;
  }

  if (estop_state != prev_estop_state) {
    Serial.print("ESTOP_PIN is now: ");
    Serial.println(estop_state == HIGH ? "HIGH" : "LOW");
    prev_estop_state = estop_state;
  }

  //  Confirm ISR handling separately
  if (estop_triggered) {
    Serial.println("[E-STOP] ISR Activated!");
    digitalWrite(LED_PIN, HIGH);
    delay(1000);
    digitalWrite(LED_PIN, LOW);
    estop_triggered = false;
  }

  delay(100);  // Sampling delay for stability
}

#endif
```

WebServer_µC_rf: Input/Output Verification Code

```
#ifdef TEST_AP_SERVER

#include <WiFi.h>
#include <WebServer.h>
#include <SPIFFS.h>

const char* ssid = "ESP32-Web";
const char* password = "TestAPESP32";
```

```
WebServer server(80);

unsigned long uploadStartTime = 0;

void handleFileUpload() {
  HTTPUpload& upload = server.upload();

  if (upload.status == UPLOAD_FILE_START) {
    Serial.printf("Upload Start: %s\n", upload.filename.c_str());
    uploadStartTime = millis();  // Start timer
    File file = SPIFFS.open("/" + upload.filename, FILE_WRITE);
    file.close();
  }
  else if (upload.status == UPLOAD_FILE_WRITE) {
    File file = SPIFFS.open("/" + upload.filename, FILE_APPEND);
    if (file) {
      file.write(upload.buf, upload.currentSize);
      file.close();
    }
  }
  else if (upload.status == UPLOAD_FILE_END) {
    unsigned long uploadDuration = millis() - uploadStartTime;  // Stop timer
    Serial.printf("Upload End: %s (%u bytes)\n", upload.filename.c_str(), upload.totalSize);
    Serial.printf("Upload Time: %lu ms (%.2f seconds)\n", uploadDuration, uploadDuration / 1000.0);
    server.send(200, "text/plain", "File Uploaded Successfully");
  }
}

void setup() {
  Serial.begin(115200);
  Serial.println("Starting ESP32 Access Point...");

  if (!SPIFFS.begin(true)) {
    Serial.println("SPIFFS Mount Failed!");
    return;
  }
  Serial.println("SPIFFS Mounted Succesfully!");
  WiFi.softAP(ssid, password);
  IPAddress IP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(IP);

  server.on("/", HTTP_GET, []() {
    server.send(200, "text/html",
```

```
    "<h2>ESP32 File Upload</h2>"
    "<form method='POST' action='/upload' enctype='multipart/form-data'>"
    "<input type='file' name='upload'><input type='submit' value='Upload'></form>");
 });

 server.on("/upload", HTTP_POST, []() {
  server.send(200);
 }, handleFileUpload);

 server.begin();
 Serial.println("HTTP server started");
}

void loop() {
 server.handleClient();
}

#endif
```

µC_Drivers_dsig: Output Verification Code

```
#define STEP_PIN `~1  // Replace with your desired GPIO pin

void setup() {
 Serial.begin(115200);
 pinMode(STEP_PIN, OUTPUT);
}

void loop() {
 static unsigned long lastPulseTime = 0;
 static int pulseCount = 0;
 unsigned long currentTime = micros();

 if (currentTime - lastPulseTime >= 1000) { // 1 kHz frequency
  digitalWrite(STEP_PIN, HIGH);
  delayMicroseconds(10); // Pulse width of 10 µs
  digitalWrite(STEP_PIN, LOW);
  lastPulseTime = currentTime;
  pulseCount++;
  Serial.print("Pulse Count: ");
  Serial.println(pulseCount);
 }
}
```

Future Recommendations

This block was a little more challenging to design as I continuously found myself down many rabbit holes while exploring the multiple interfaces of the block and finding areas of improvement or exploring alternative ideas that could help the project. Sometimes, however, I did feel like I spent a disproportionate amount of time researching, to diminishing returns. I would tell myself from the start of this term to keep a list of priorities so that I don't lose focus when researching, and to journal my progress in order to keep my thoughts ordered, and smooth the process of researching, writing the document and developing my block. This time compared to the first block I spent a lot more time in verification, and on datasheets trying to find the most relevant properties that I could test, and ideating validation strategies I can use for my verification. I think the learning from this block I can apply elsewhere is really just the need for journaling, and keeping good track of my ideas, to smoothly integrate them into my work.

References

[11] Espressif Systems, "ESP32 Series Datasheet," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf [Accessed: Apr. 20, 2025]

[12] Espressif Systems, "ESP-WROOM-32 Datasheet." https://www.mouser.com/datasheet/2/891/esp-wroom-32_datasheet_en-1223836.pdf?srsltid=AfmBOoq51T8_Wn243PBMdjcEAsQfkJYYEVY2hw_7I4RFvrVibQ8lw0lm](https://www.mouser.com/datasheet/2/891/esp-wroom-32_datasheet_en-1223836.pdf?srsltid=AfmBOoq51T8_Wn243PBMdjcEAsQfkJYYEVY2hw_7I4RFvrVibQ8lw0lm)

[13] ON Semiconductor, "NCP1117: 1.0 A Low Dropout Positive Fixed and Adjustable Voltage Regulators." [Online]. Available: https://www.onsemi.com/pdf/datasheet/ncp1117-d.pdf

[14] Arduino Kit Project, "Learn DC Motor Control with ESP32 and L298N Driver," Arduino Kit Project, [Online]. Available: https://arduinokitproject.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/ [Accessed: Apr. 20, 2025].

[15] "Wi-Fi Standards." Ting Help Center. https://help.ting.com/internet-articles/wi-fi-standards (accessed May 08, 2025).

[16] "How to use SPIFFS for an ESP32 File System [Beginner Guide + Code]." Programming Electronics Academy. https://www.programmingelectronics.com/spiffs-esp32/ (accessed May 09, 2025).

[17] STMicroelectronics, "L298 - Dual full-bridge driver," Datasheet DS0218, Rev. 5, Oct. 2023. [Online]. Available: https://www.st.com/resource/en/datasheet/l298.pdf. [Accessed: May 11, 2025].

[18] PBC Linear, "Stepper Motor Support Data Sheet," [Online]. Available: https://pages.pbclinear.com/rs/909-BFY-775/images/Data-Sheet-Stepper-Motor-Support.pdf. [Accessed: May 11, 2025].

[19] JOY-IT, "NEMA 17 Stepper motor - NEMA17-06," [Online]. Available: https://joy-it.net/en/products/NEMA17-06 [Accessed: May 11, 2025]