IMPERIAL

Second Year Group Project

Imperial College London

Department of Electrical and Electronic Engineering

TED - Balance Bot

Sara Chehab
02300594
sc2922@ic.ac.uk

Varun Sangtani
02209157
vs822@ic.ac.uk

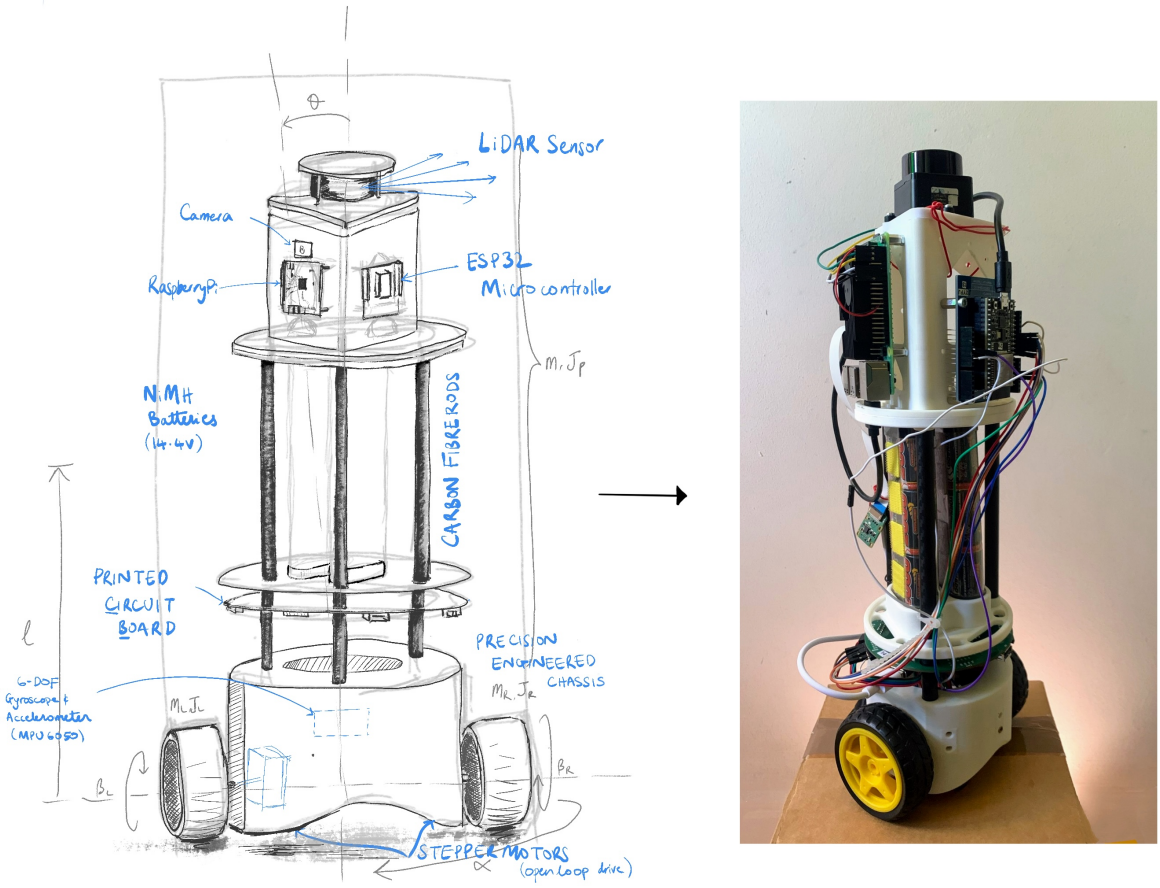
Veer Tandon
02224814
vt222@ic.ac.uk

Kevin Lau
02240867
khl22@ic.ac.uk

William Huynh
02271593
wh1022@ic.ac.uk

Sanjit Raman
02032046
ssr321@ic.ac.uk

June 2024
Word count: 9,016


LiDAR Sensor
Camera
ESP32
Micro controller
RaspberryPi
M_P, J_P
Ni MH
Batteries
(14.4V)
CARBON FIBRE RODS
PRINTED
CIRCUIT
BOARD
l
PRECISION
ENGINEERED
CHASSIS
M_R, J_R
6-DOF
Gyroscope &
Accelerometer
(MPU 6050)
$M_L J_L$
B_L
B_R
STEPPER MOTORS
(open loop drive)

Abstract

The second-year balance bot design project focuses on creating an autonomous two-wheeled robot with an open-ended application. We propose the Tactical Environment Detector (TED), a robot capable of physical incident detection through dynamic environment mapping and automated exploration. Ideal environments for the robot include well-lit spaces such as warehouses and livestock farms.

ESP32 and Raspberry Pi modules are used to process navigation instructions, run continuous loops to facilitate balancing, and maintain constant communication with a cloud server. Visual SLAM techniques are implemented locally using a LiDAR sensor for accurate mapping. The design includes a power management system to convey relevant battery information to the user.

The robot demonstrates effective navigation, quick mapping, and balance, and showcases the integration of mechanics, vision, communication, and control subsystems.

Contents

1 Introduction

This report will highlight the group's development of TED over the past 4 weeks. The project hinges on the following brief:

> *"The aim of this project is to build a demonstrator robot that can balance on two wheels. You should choose an application for the robot that will demonstrate interaction with human users or the environment. A robot platform is provided, which contains wheels, motors, batteries, power electronics, and mounting points for embedded computing platforms."*

A demonstrator application was chosen that aims to minimise labour-intensive tasks that can be automated, improving efficiency and productivity:

TED is a scalable incident management platform that discovers incidents in indoor environments such as warehouses. These incidents include overturned boxes, spillage of hazardous substances and equipment malfunctions. A valid demonstration arena would embody the following characteristics of our chosen environment:

1. **Clustered areas**

   Logistics warehouses contain areas where various boxes are placed in small clusters, challenging the robot's navigation abilities.

2. **GPS-limited environment**

   The robot has to operate within an indoor warehouse, which is often GPS-limited.

3. **Dynamic environment**

   Dynamic obstacles, such as forklifts, operate in the warehouse. The robot has to adapt to these changes and employ obstacle avoidance.

Design Specifications

The finalised demonstration arena is shown in Figure 1 and was constructed with recycled cardboard. An incident is represented by a green-coloured crate. Expanding on the project brief concerning our chosen application, the bot must meet the following design specifications:

1. Be able to map an unknown environment dynamically without GPS, suggesting the usage of Simultaneous Localisation and Mapping (SLAM).
2. Identify incidents (characterized by a green-coloured crate) and report this to the user.
3. Report remaining battery life and power consumption to the user.
4. Balance on two wheels.
5. Move fast enough to scan the arena in under 10 minutes.
6. Explore and navigate the arena autonomously.
7. Aggregate data is presented using an intuitive user interface.
8. Have a custom head unit that suits its purpose as a demonstrator.

Figure 1: Demonstration Arena

2 Planning

2.1 Schedule

To effectively organize our project, we created a rough plan of what needed to be done, as shown in the Gaant chart in Figure 2.

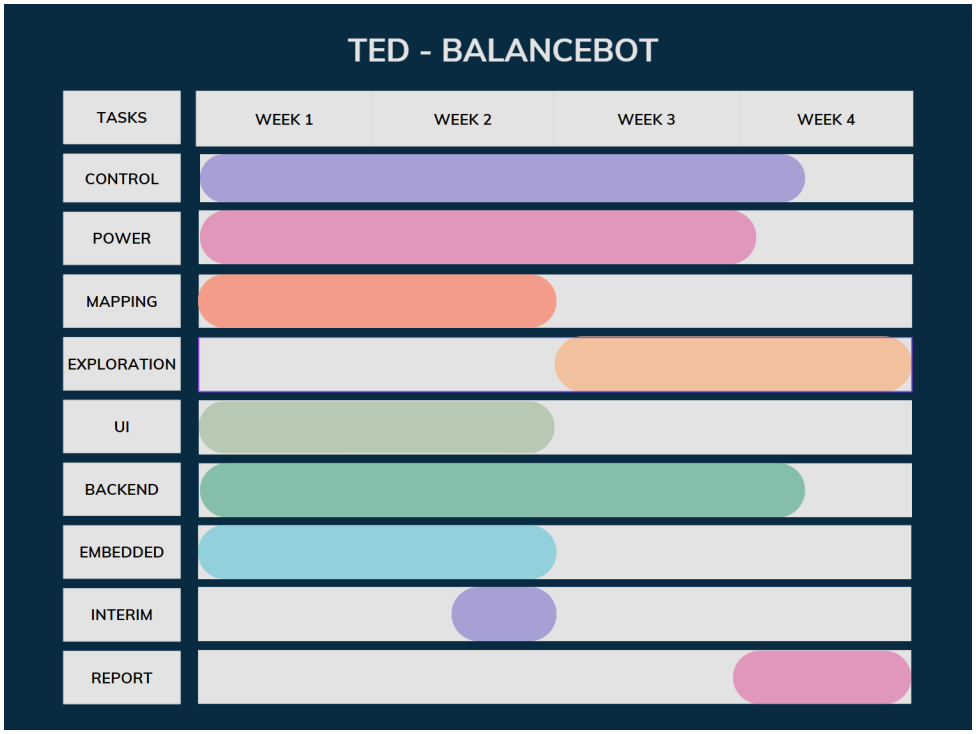


Figure 2: Gaant Chart

Throughout the project, every consultation opportunity was taken to refine our ideas. Regular team meetings were also held to raise concerns and inform other group members of progress.

We adopted an agile framework as a team, which involved breaking the project into phases and emphasising continuous improvement [1]. These phases, also known as *sprints*, are weekly cycles in which we plan our activities based on the outcomes of our previous sprints [2]. Details of the weekly sprints are shown in Table 1.

2.2 System Decomposition and Testing Methods

Using the overall system design criteria, we identified necessary subsystems and how they should be tested, summarised in Table 2.

A testing methodology is laid out in accordance with the agile framework to diminish any chances of error, shown in 3.

2.3 Task Distribution

Roles were allocated based on individual team member skills and preferences. Figure 4 shows the division of responsibility among team members.

Sprint	Duration	Goals/Activities
1	Week 0	• Initial design and requirement gathering • Research and selection of hardware components • Initial setup of development environment • Basic robot framework design • Basic self-balancing algorithm prototype
2	Week 1	• Inner loop of PID tuning for control • MQTT Communication establishment • UI design • Initial power tracking development • Research into SLAM (Simultaneous Localization and Mapping)
3	Week 2	• Interim presentation • Outer loop control • Database design and querying • Research into mutex for embedded systems • Refinement power circuit • Map building with SLAM
4	Week 3	• Turning controls • Autonomous algorithm development • Movement control from UI • Soldering and testing of power components
5	Week 4	• Integration of all components • Report preparation • Demonstration, preparation and execution • Hooking up map to UI

Table 1: Sprint Plan for Autonomous Robot Project

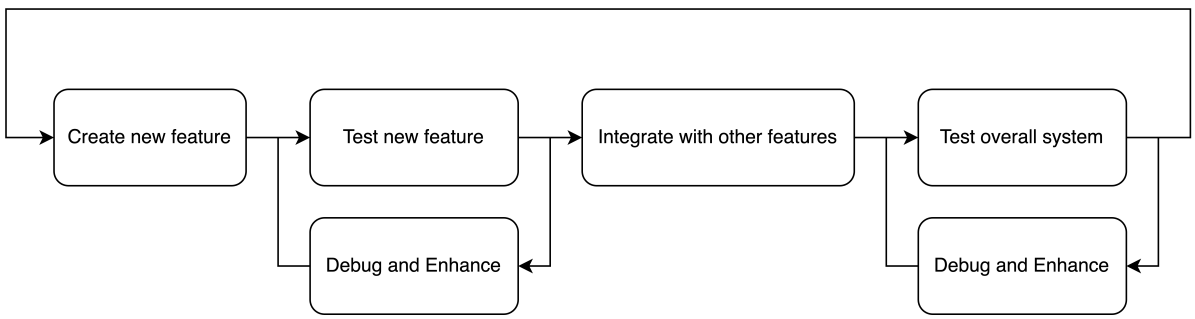


Figure 3: Testing methodology

Section	Subsystem	High-Level Criteria	Testing Methods
4	Balancing and Movement	• Stabilises the bot • Facilitates movement in 2 dimensions	• Test accuracy of all sensor measurements • Ensure robot speed and direction matches the target values
5	Battery and Power	• Measure instantaneous power usage • Measure energy consumption	• Observing accuracy of power consumption and battery percentage
6	Mapping and Exploration	• Simultaneously map and keep track of robot position in dynamic environment • Autonomously explore unknown environment • Identify incidents in environment • Send report (map and image) of incident to web server	• Map building algorithm tuning • Simulation environment testing (Gazebo sim) • Navigation crash avoidance • Pure pursuit look-ahead distance tuning
7	Software System	• Simultaneously map and keep track of robot position in dynamic environment • Send/recieve information with a low latency • Save data for future querying	• Checking connection for reliability, correctness and latency • Database for correctness and latency • Website functionalities for low-latency, dynamic (live) updates
8	Embedded	• Integrate hardware and software	• Automated tests using the PlatformIO test framework

Table 2: Subsystem Criteria and Testing

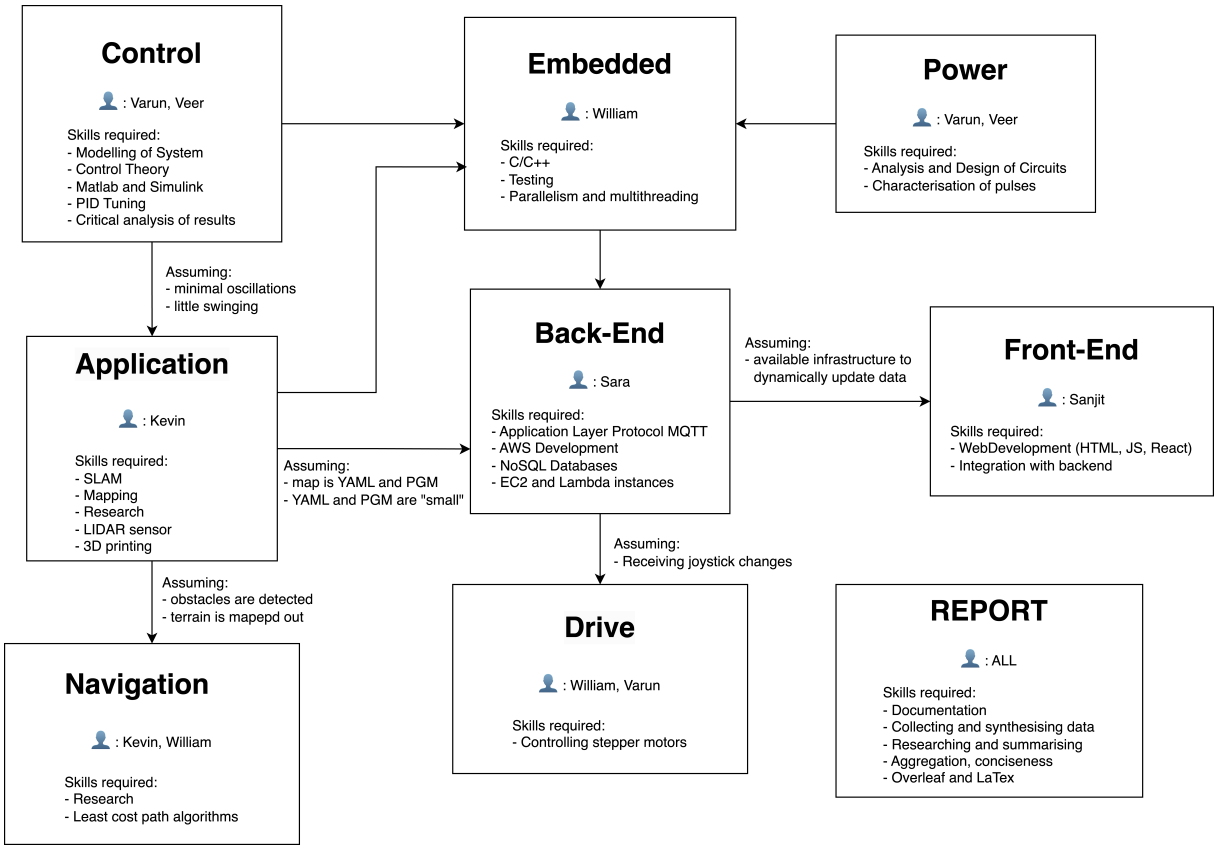


Figure 4: Role allocation within team

3 High-Level System Overview

3.1 System Architecture

The high-level diagram of the bot, with modes of transmission, is shown in Figure 5.

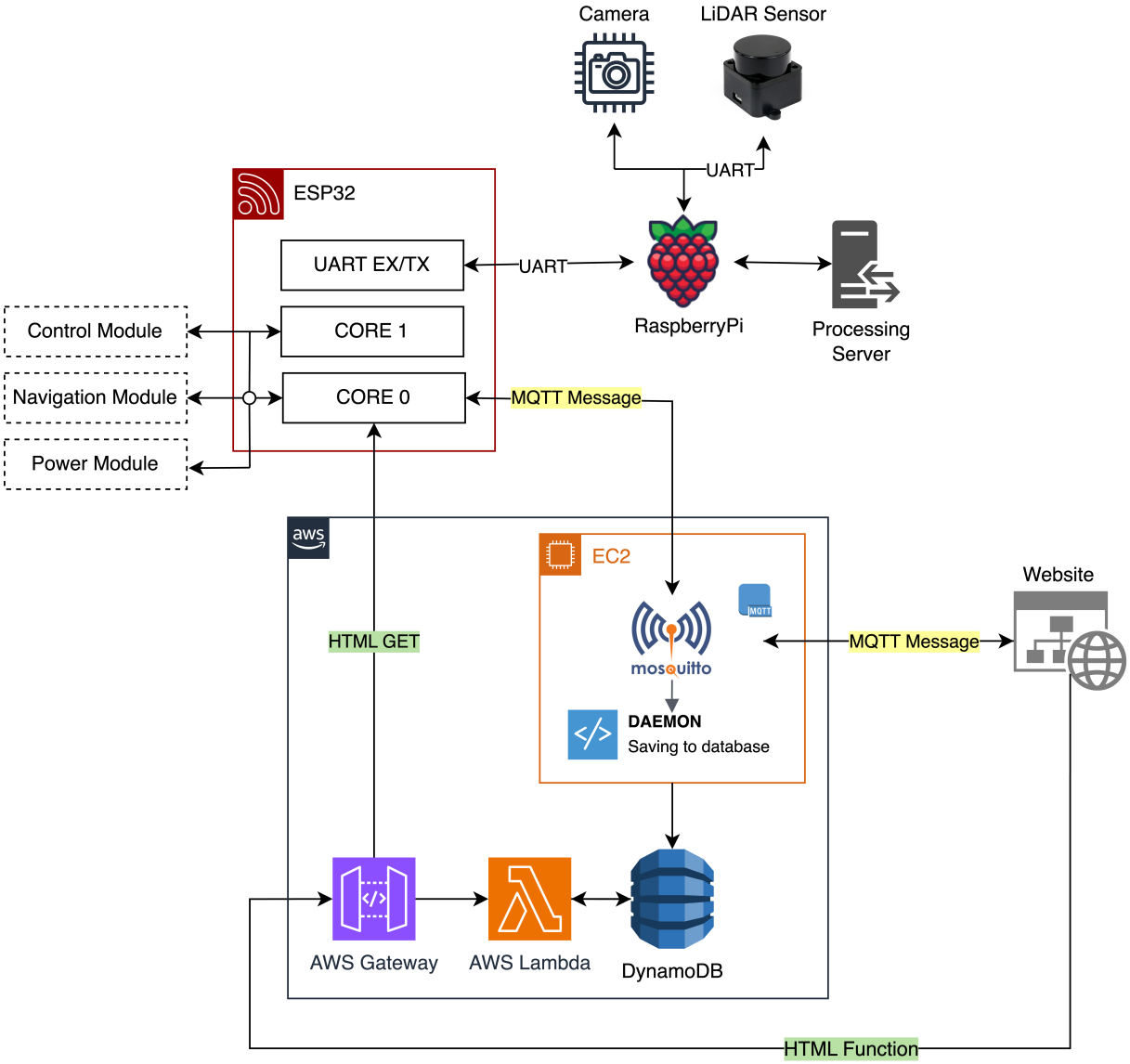


Figure 5: Overall System Architecture

Note that 2 devices are connected to the Raspberry Pi via UART. This is made possible by using GPIO pins 14, 15, 5V and GND [3].

3.2 Head Unit

As mentioned above, a LiDAR is used for sensing and must be mounted on the robot. The head unit was designed to be minimalist, in that it makes as few changes to the original design as possible, using every side to fit the various sub-modules. Figure 6 shows the head unit, featuring the power stripboard (on the hidden face), ESP32, Raspberry Pi, Camera and LiDAR sensor.

Note that the LiDAR's centre of mass was aligned with the robot's centre of mass to minimise the energy needed for balancing.

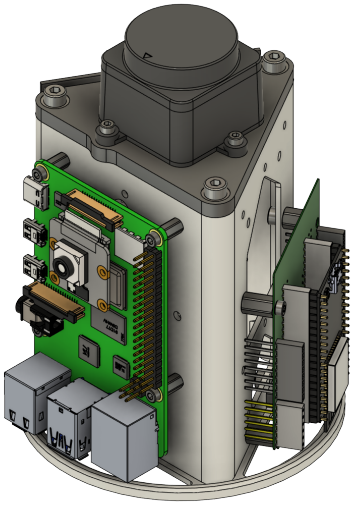


Figure 6: Diagram of the custom-designed head unit

3.3 Schematic

Figure 7 shows the high-level schematic of all components of the robot

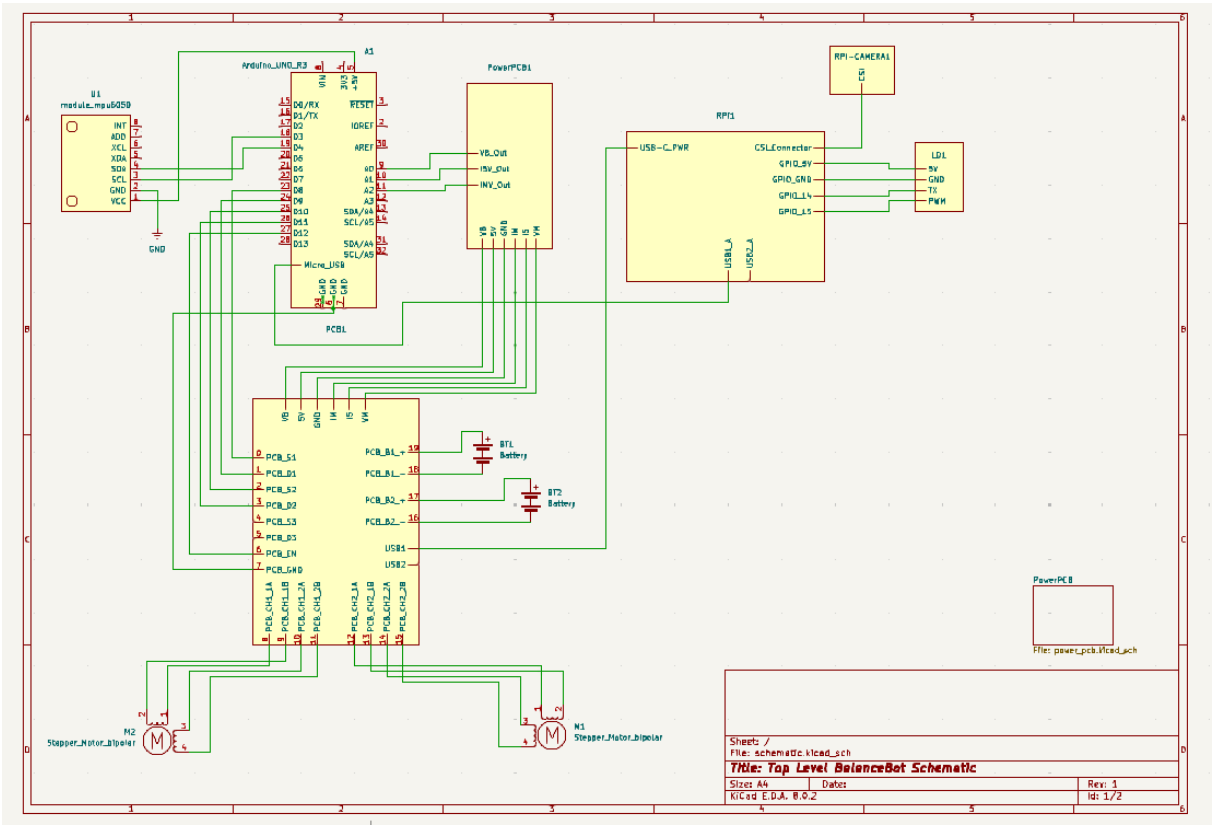


Figure 7: High-Level Schematic

4 Balancing and Movement

4.1 Introduction and Design Criteria

A two-wheeled robot presents a nonlinear system with no stable equilibria at which it can be operated. Therefore, a control system must be designed to both stabilise the robot and facilitate its movement. Movement commands in the form of velocity and yaw rate are provided either through autonomous action or manual control. A successful control algorithm would satisfy the following design criteria:

(a) Balance the robot in an upright position with no drift in position.

(b) Smoothly accelerate to and maintain a desired velocity with minimal steady-state error.

(c) Accurately track yaw according to a desired yaw rate in both static and dynamic conditions.

(d) Be able to operate with reasonable disturbance (external forces) and uneven surfaces.

Achievement of these design criteria will ensure that the robot is capable of steering and movement in real-world environments.

4.2 Insights from System Dynamics

A simplistic model of the two-wheeled robot is an inverted pendulum on a cart. While this model does not fully represent the robot, understanding its dynamics may inform PID tuning methods.

Figure 8 illustrates the proposed situation. The pendulum consists of a point mass, with mass m, at the end of a rigid and massless rod of length l. The point mass represents the mass of the robot body, and l is the distance to the centre of mass. Here, we assume the centre of mass of the robot lies on its vertical axis. A cart of mass M depicts the robot wheels. θ is the tilt angle.

In a configuration where the pivot is fixed in space, the following equation depicts tilt dynamics [4]:

$$\ddot{\theta} = \frac{g}{l} sin(\theta) \tag{1}$$

The inverted pendulum accelerates away from the vertical at a rate inversely proportional to l. This state equation is linear for small tilt angles (less than 15 deg). To ensure operation under linear dynamics, emphasis was placed on tilt measurement accuracy and steady-state error minimisation during tuning.

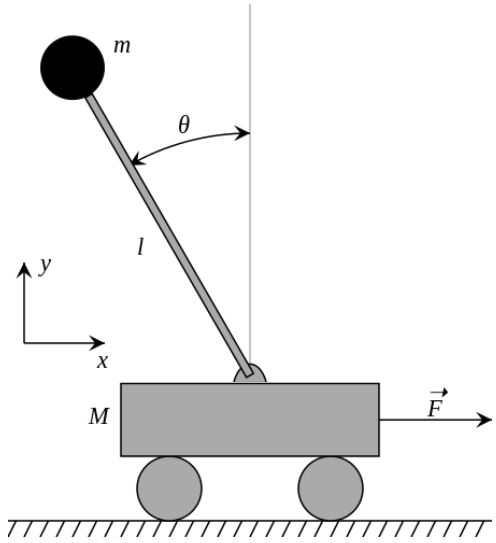


Figure 8: Inverted pendulum schematic [5]

4.3 Control Architecture

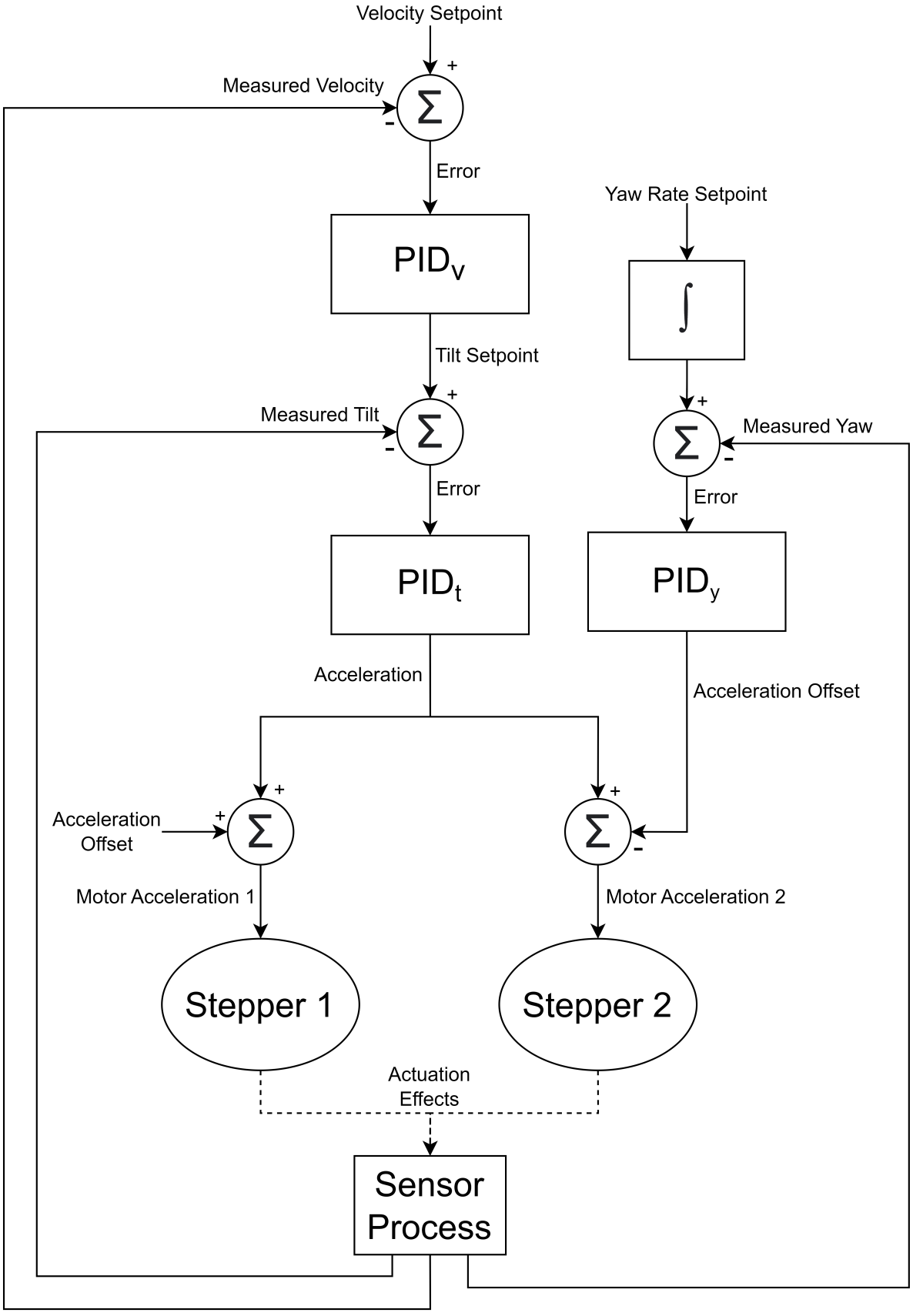


Figure 9: Control Scheme Block Diagram

An overview of the final control scheme is illustrated in Figure 9. A cascaded loop structure controls velocity; the outer loop sets a desired tilt angle for the inner loop to track. An external loop handles yaw control. Table 3 summarises each loop's operation.

Control Loop	**Purpose**	**Mode of Action**
Tilt (Inner)	Match robot tilt θ with setpoint from velocity loop	Adjust motor accelerations together
Velocity (Outer)	Match robot velocity v with setpoint from user or autonomous action	Adjust tilt setpoint
Yaw	Match yaw rate with setpoint from user or autonomous action	Adjust motor accelerations individually

Table 3: Comparison of Voltage Measurement Methods

Table 4 details the finalised control parameters. Inner and outer loop frequencies were chosen such that tilt and velocity dynamics are sufficiently decoupled. This decoupling was imperative to the success of the cascaded control system. Since the yaw and tilt control loops both affect motor acceleration, yaw control was chosen to run at half the frequency to ensure balancing action occurs more frequently than rotation.

Justification for the scheme and details of the sensor process are presented in the following subsections.

Control Loop	Frequency $[Hz]$	Proportional Gain (k_p)	Integral Gain (k_i)	Derivative Gain (k_d)
Tilt (Inner)	100	45	0	0
Velocity (Outer)	5	0.5	0.1	0.02
Yaw (External)	50	3	0	5

Table 4: Control Scheme Parameters

4.4 PID Implementation

All three PID control loops must be implemented in discrete-time. A controller relates an error term, which is calculated as the difference between the desired setpoint and measured value, to its output. This behaviour is characterised in continuous-time by:

$$v(t) = k_p e(t) + k_i \int_0^\tau e(\tau) d\tau + k_d \frac{de_f(t)}{dt} \tag{2}$$

where $e(t)$ is the error term and $v(t)$ is the controller output. The derivative term is calculated via a low-pass filtered error, to limit noise. This scheme was translated to code in Listing 1 [6].

```
error = setpoint - measured_value;

// using rectangle approximation
integral += error/dt;

// using finite difference approximation
unfiltered_derivative = (error - previous_error) / dt;

// filtering with low-pass filter
derivative = (tau / (tau + dt)) * previous_derivative
                    + (dt / (tau + dt)) * unfiltered_derivative;

output = kp * error + ki * integral + kd * derivative;

previous_error = error;
previous_derivative = derivative;
```

Listing 1: Calculating derivative term

Without an accurate model of the robot, the entire control system must be tuned through trial-and-error. Research was conducted into standard tuning approaches such as the Ziegler-Nichols method. However, using this method requires a system that is able to oscillate indefinitely (or for a substantial amount of time) [7]. This is unfortunately not the case for the two-wheeled robot.

To streamline PID tuning, a dedicated server was created to change gain parameters on the fly (Figure 10).

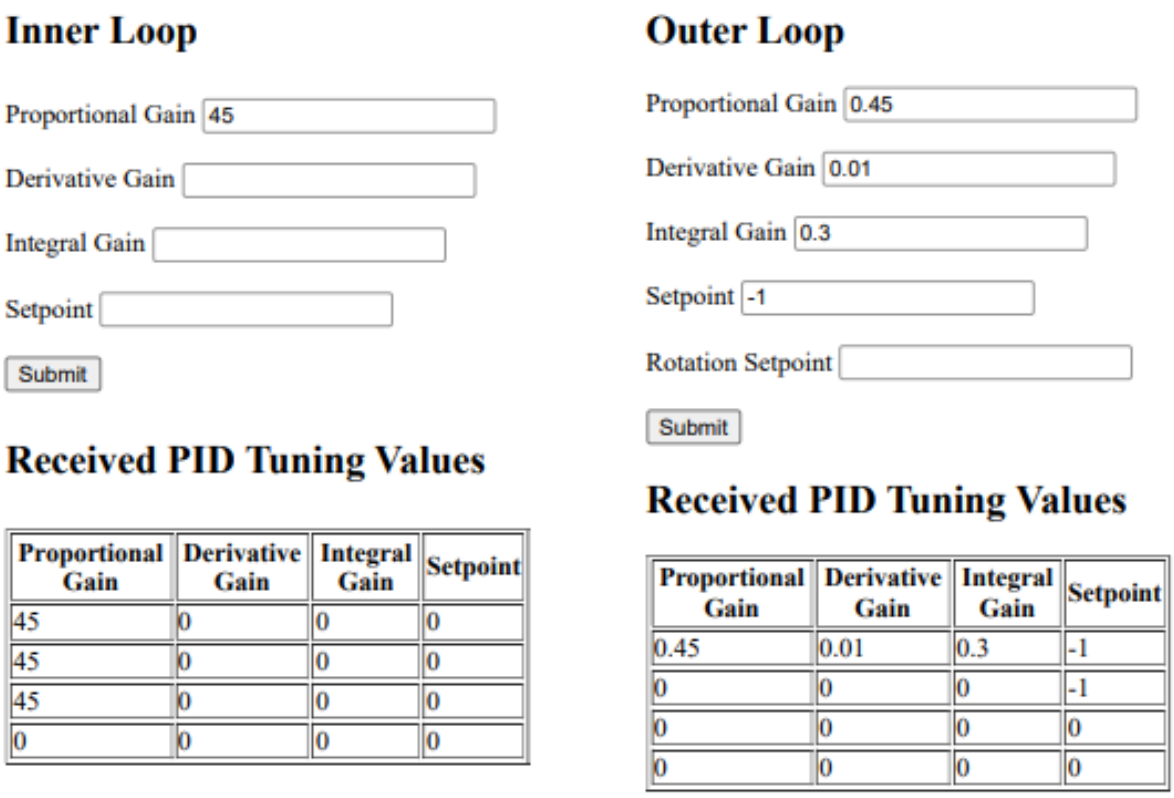
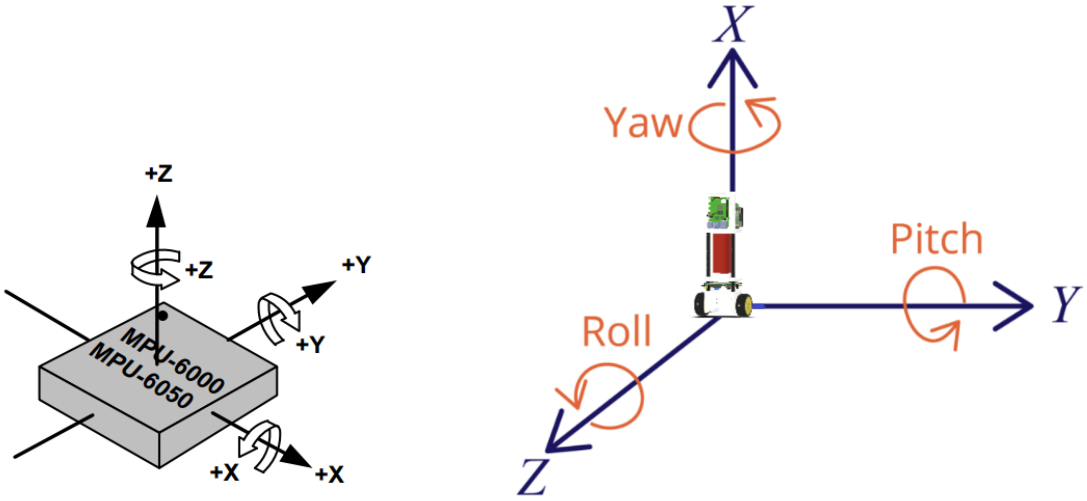


Figure 10: Website used for PID tuning

4.5 Tilt Control

4.5.1 Measuring Tilt Angle

Response of the control system heavily relies on access to precise sensor data. Tilt (pitch) measurement is achieved using the MPU-6050, a 3-axis accelerometer and gyroscope module (Figure 11a). The sensor is mounted in the base unit of the robot chassis for strong mechanical coupling to the wheels. On its internal mounting point, the sensor's x-axis points vertically while the z-axis points in the direction of motion (Figure 11b). A measurement library with simple commands to retrieve sensor data was provided with the project starter code.



(a) Sensor axes

(b) Orientation of the sensor axes

Figure 11: MPU-6050 module

Before using any sensor readings, the MPU-6050 must be calibrated [8]. To do this, the robot is started in the upright position on the provided stand (0° tilt). The expected results are:

- $a_x = 9.81\, ms^{-2}$, $a_y = 0\, ms^{-2}$, $a_z = 0\, ms^{-2}$
- $\omega_x = 0\, rads^{-1}$, $\omega_y = 0\, rads^{-1}$, $\omega_z = 0\, rads^{-1}$

where $a_{x,y,z}$ is acceleration along an axis, and $\omega_{x,y,z}$ is rate of change of angle about an axis (gyroscope reading). A calibration function takes the mean of 200 sensor readings and compares this to the expected values so that the offset is accounted for in every subsequent measurement. This runs every time the ESP32 is restarted.

With a fully calibrated sensor, the accelerometer can be used to calculate tilt according to:

$$\theta_a = atan\left(\frac{a_z}{\sqrt{{a_x}^2 + {a_y}^2}}\right) \tag{3}$$

A small-angle approximation is used to reduce computational load since we do not expect tilt to exceed 15° (0.26 rad):

```
acceleration_angle = (accZ)/sqrt(pow(accY, 2)+pow(accX, 2));
```

Listing 2: Small angle approximation

The accelerometer is unable to distinguish between acceleration due to gravity and acceleration due to driving force from the motors. Therefore, a tilt angle calculated simply using the accelerometer is only accurate at constant velocity i.e. accurate in the long-term. To overcome short-term error, gyroscope readings about the y-axis must also be used [9].

Since the gyroscope measures the rate of change of tilt, it must be integrated. Tilt is given by:

$$\theta_g = \int_0^t \omega_y(\tau) d\tau \tag{4}$$

For the digital implementation, we use a rectangle approximation ($T_s = 10ms$):

$$\theta_g(kT_S) = \theta_g((k-1)T_s) + T_s\omega_y(kT_s) \tag{5}$$

In the z-domain:

$$\theta_g(z) = \frac{T_s z}{z-1}\omega_y(z) \tag{6}$$

While θ_g is more accurate that θ_a over a short-time frame, the gyroscope's susceptibility to low-frequency noise causes drift in the measurement over time. The situation is summarised as follows:

- We trust the long-term average of θ_a but not any fast changes.
- We have confidence in short-term changes of θ_g but not in the long-term average.

To combine the merits of both measurements and eliminate the drawbacks, sensor fusion is implemented through a complementary filter as:

$$\theta_n(kT_s) = (1-C)\theta_a(kT_s) + C(\theta_g((k-1)T_s) + T_s\omega_y(kT_s)) \tag{7}$$

where C is the filter coefficient. In the z-domain:

$$\theta_n(z) = \frac{(1-C)z}{z-C}\theta_a(z) + \frac{T_s C z}{z-C}\omega_y(z) \tag{8}$$

θ_n is denoted as `robot_angle` within the control algorithm code. To find a suitable value for C, values between 0.9 and 1.0 were evaluated by measuring `robot_angle` while moving between $+15°$and $-15°$(measured with a protractor). A final value of $C = 0.98$ was chosen, with testing in Figure 12.

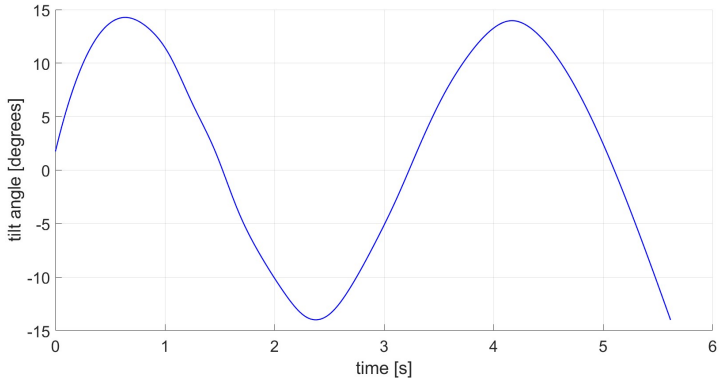


Figure 12: Tilt accuracy test with $C = 0.98$

4.5.2 Inner Loop Design Considerations and Tuning

Since 0° tilt was defined on the robot stand, the first step in tuning was to measure the tilt at which the robot naturally balances (denoted by `balance_setpoint` in the code). During the tuning process we had *balance_setpoint* $= 0.7°$. PID gains were tuned by trial-and-error such that the robot balanced around this setpoint.

Before fine-tuning PID gains, we explored the impact that motor microstepping had on the system. We found that the speed of the motors decreased as microstepping was introduced, limiting their potential

driving force. Operating in full-step mode yielded the best response. The three-way configuration switch setting is in table 5.

MS1	MS2	MS3
LOW	LOW	LOW

Table 5: Switch settings for microstepping [10]

After tuning PID gain, a surprising result was that the controller performed best with k_p only. Any increase in k_d worsened oscillations. One disadvantage of the PID topology discussed above is a 'derivative kick' effect that arises from sudden disturbances [11]. To mitigate this, the derivative of the measurement instead of the error was used $\left(k_d \frac{d\theta_n(t)}{dt}\right)$. Nonetheless, introducing derivative gain still worsened performance. It was then agreed that a proportional gain would be sufficient; k_p is chosen at 45 after the testing detailed in the section below.

4.5.3 Testing

Since manual tuning was done through trial and error, a framework was required to compare performance of the PID values which successfully balanced the robot. This framework involved recording the response of the system to two control inputs. The response of the final parameters only is shown.

Figure 13 depicts the response to a train of impulses while Figure 14 shows the response to changing step inputs. From these, it can be asserted that the system performs adequately:

- Tracking around the balancing point is consistent with oscillation amplitude less than 0.2°.
- Tilt responds very quickly to changes in setpoint, overshooting by only 0.5°.

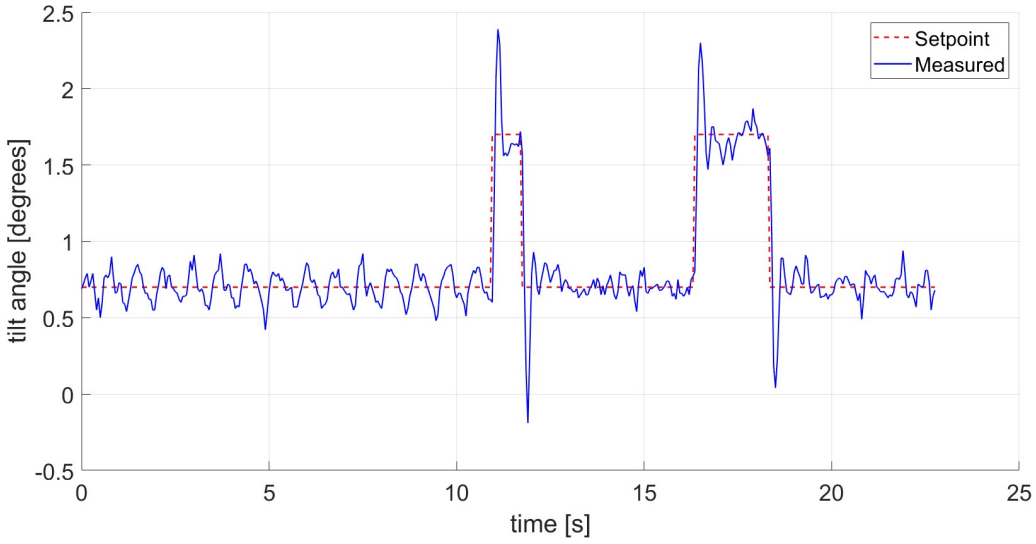


Figure 13: Inner loop response to a train of two impulses ($k_p = 45$, $balancing\_setpoint = 0.7°$)

Our design opts for quicker response at the expense of higher overshoot. This is because the testing depicts a worst-case situation with 1° changes in tilt setpoint. That situation can be avoided by exploiting speed setpoint ramping. This will be further discussed in the following section.

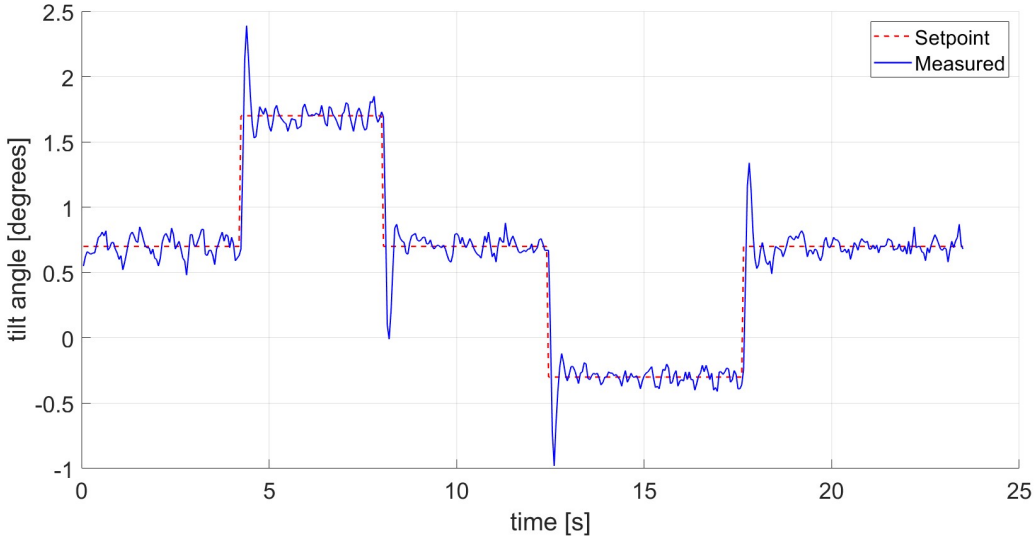


Figure 14: Inner loop response to step inputs ($k_p = 45$, $balancing\_setpoint = 0.7°$)

4.6 Velocity Control

4.6.1 Measuring Robot Velocity

It is imperative for velocity control that a measure of motor speed accurately depicts the current motion of the robot. The provided library for stepper motor control includes a `getSpeedRad()` function which returns the current speed of a stepper motor in $rad\,s^{-1}$. Figure 15 shows typical unfiltered readings of motor speed in the balancing conditions. In these conditions, the robot has zero displacement in the long-term and so robot velocity should be zero also. The figure, however, shows that motor speed is non-zero due to the constant acceleration and deceleration to maintain balance.

Motor speed readings are low-pass filtered using an exponential moving average:

```
filtered_value = alpha * ((step1.getSpeedRad()-step2.getSpeedRad())/2)
                 + (1 - alpha) * filtered_value;
```

Listing 3: Exponential moving average

where *alpha* is the smoothing factor. This averaging occurs every time the inner loop is run ($100Hz$). $alpha = 0.05$ was chosen based on this frequency, giving the filtered response shown in the figure. Filtered motor speed varies significantly less while still responding quickly to changes.

4.6.2 Outer Loop Design Considerations and Tuning

An inner-loop setpoint can now be calculated using the previously calculated speed measurement. Motor speed was limited to -2 to 2 to suit mapping purposes and limit tilt.

In preparation for the PID tuning process, the inner loop testing results were revisited. It was then established that sudden tilt setpoint changes caused overshoots in the response. This behaviour was dampened by implementing an exponential speed ramping function, as can be seen in Listing4.

```
beta = 1 - exp(-LOOP2_INTERVAL / time_constant);
current_speed_setpoint += beta * (speed_setpoint - current_speed_setpoint);
```

Listing 4: Exponential speed ramping function

The above code runs within the outer loop, smoothing transitions to new speed setpoint. Therefore,

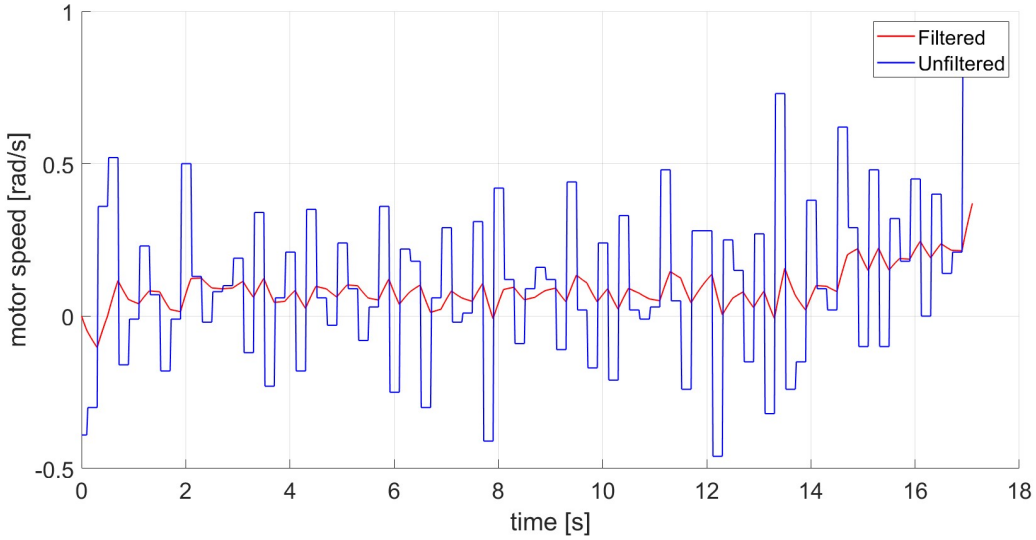


Figure 15: Time evolution of motor speed while balancing

spikes in the tilt setpoint are avoided. $time\_constant = 2000$ was chosen such that ramping from 0 to 2 took under 3 seconds.

PID gains were tuned through trial-and-error. k_p was increased by itself until it effectively started motion in the desired direction. Once this was achieved, k_i was introduced to eliminate steady-state error. Finally, a small k_d helped to dampen oscillations. The finalised parameters are $k_p = 0.55, k_i = 0.12, k_d = 0.02$.

4.6.3 Testing

A series of tests were devised on the provided arena to assess the performance of the outer loop on an uneven surface:

1. End-to-end movement at constant velocity ($speed\_setpoint = 1.5$).
2. Movement over 2 metres at constant velocity with human interference.
3. End-to-end movement with a varying setpoint.

Tests 1 and 3 revealed that the control loop successfully moved the robot in the correct direction but exhibited start-stop behaviour. With the robot unable to maintain a consistent velocity, we decided to impose the following limits on the output tilt setpoint:

Speed Setpoint	Min. Tilt Setpoint	Max. Tilt Setpoint
$> \mathbf{0}$	$balancing\_setpoint - 5$	$balancing\_setpoint + 1$
$< \mathbf{0}$	$balancing\_setpoint - 1$	$balancing\_setpoint + 5$

These limits help in managing the transient response of the system by avoiding excessive overshoot or undershoot. This change completely corrected the issue.

Test 2 involved human disturbances e.g. small pushes. The control loop was able to reject these but did not recover after the robot was picked up and moved. To prevent these integrator windups in the case of such user intervention, the integral term is set to zero soon as the outer loop's k_i is set to zero, as can be seen in Listing 5.

Therefore, the user is able to set $k_i = 0$ on the UI in the case of an intervention. Robot control is then recoverable without having to restart the entire algorithm.

With the above changes, the robot was indeed able to smoothly accelerate to and maintain a desired velocity with minimal steady-state error.

```
if (ki_o == 0){
    integral_speed = 0;
}
```

Listing 5: Prevent integral wind-up

4.7 Yaw Control

4.7.1 Measuring Yaw Angle

Gyroscope readings about the x-axis were used to find yaw rate. Yaw angle control requires integration of this measurement, as can be seen in Listing 6.

```
rotation_rate = g.gyro.x-gyroXoffset;
rotation_angle = rotation_rate * (LOOP3_INTERVAL / 1000) + rotation_angle;
```

Listing 6: Yaw angle

Yaw angle control was chosen over yaw rate control for the following reasons:

1. Yaw rate control may lead to drift, reducing orientation precision.
2. Corrections for any deviation is much more intuitive.
3. Although the current control scheme receives a yaw rate setpoint, yaw angle control opens the door to precise orientation tasks in the future.

4.7.2 Loop Design Considerations and Tuning

The control loop receives a yaw rate setpoint. This is integrated to obtain a target angle: 7.

```
rotation_target_angle = rotation_target_angle +
                        (rotation_setpoint*0.4*LOOP3_INTERVAL)/1000;
```

Listing 7: Rotation Target

A factor of 0.4 is included so that a setpoint of 1 corresponds to a desired yaw rate of $45°/s$. From here, the error term is used to calculate an *acceleration offset*. This offset is added to one motor and subtracted from the other, thus facilitating yaw.

PID gains were tuned for a critically damped response to prevent any over-steering that would affect the performance of sensors used for mapping. The finalised PID values are $k_p = 3, k_i = 0, k_d = 5$.

4.7.3 Testing

Consecutive step inputs were used to evaluate the performance of the yaw control loop. The response of the finalised loop is given in Figure 16. These results show that the loop is successful in smoothly tracking a desired yaw angle with critically damped behaviour.

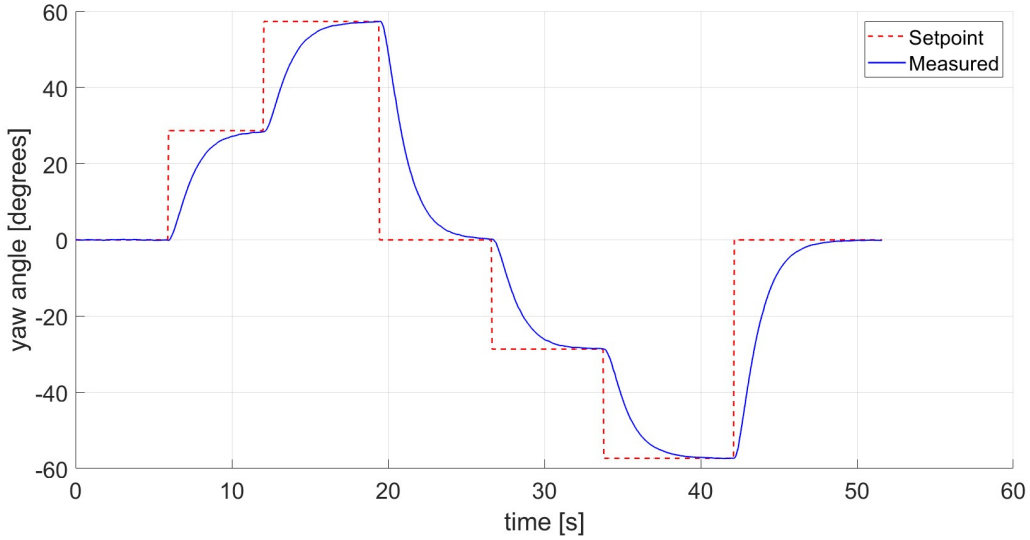


Figure 16: Yaw loop response to step inputs ($k_p = 3, k_i = 0, k_d = 5$)

4.8 Subsystem Testing and Evaluation

The first subsystem test involved manually controlling the robot over a web server using 'WASD' keyboard inputs to provide velocity and yaw rate setpoints. The robot was tested for 10 minutes with five different sets of batteries to observe any performance impacts.

The control algorithm performed adequately, meeting the specifications outlined. Control in two dimensions was intuitive and responsive, and the motors did not overheat, eliminating concerns of motor driver shutdown. However, yaw measurements drifted over time due to imperfect readings from the MPU-6050's gyroscope. Future improvements could include using a 9-axis sensor like the MPU-9250 for precise yaw calibration.

The second test, conducted by the system integration team, evaluated computational load by running the control algorithm alongside other subsystems. The loop execution times were measured as follows:

- Tilt (Inner) Loop: 8.3ms
- Velocity (Outer) Loop: 5.4ms
- Yaw Loop: 4.0ms

The inner loop's 10ms period with 8.3ms execution time leaves only 17% idle time per cycle, indicating that system priorities must be carefully managed to ensure stability, which will be expanded upon in the embedded section

While the control algorithm meets all requirements and operates successfully with other subsystems, formulating a precise model of the robot could enhance dynamic modeling and allow for advanced control algorithms like LQR and model predictive control (MPC). This highlights a road-map for future work.

FMEA Analysis of Subsystem

We conducted an FMEA of the control subsystem to identify potential risks and implement mitigation strategies. For detailed findings and recommendations, please refer to Appendix C.

5 Battery and Power

5.1 Brief

Battery and power status are essential for communicating with the user. The robot dissipates power through motor action and powering the microcontroller/processor modules. A successful subsystem achieves the following aims:

1. Accurately calculates instantaneous power usage and the energy consumed.
2. Conveys a reasonably precise battery percentage that predicts when the motors and other electronics will fail.
3. Implements safety features to avoid over-voltage situations.

5.2 Power Consumption

5.2.1 Motor vs. 5V USB

As outlined above, power consumption is broken down into power dissipated through motor action and the ESP32 and Raspberry Pi modules. The power consumed can be calculated by multiplying the current flow and the channel voltage. Power consumption is given by:

$$P_M = I_M \times V_{BAT} \quad \& \quad P_{5V} = I_5 \times V_5 \tag{9}$$

where I_M and I_5 are the currents drawn by motors and USB modules, respectively. V_{BAT} is the battery voltage, and V_5 is the voltage measured from the 5V pin.

The IM+, VM, I5+, and 5V pins on the power PCB provide the voltages across two $10m\Omega$ shunt resistors, thus giving current draw as:

$$I_M = \frac{VM - IM+}{10m\Omega} \quad \& \quad I_5 = \frac{5V - I5+}{10m\Omega} \tag{10}$$

Measuring IM+ and VM while balancing shows a similar magnitude to the battery voltage (Figure 17); their difference is between 20mV and 9mV in regular operation. The voltages vary frequently due to inconsistent motor speeds as the robot balances. To combat this, a digital or analogue filter must be implemented.

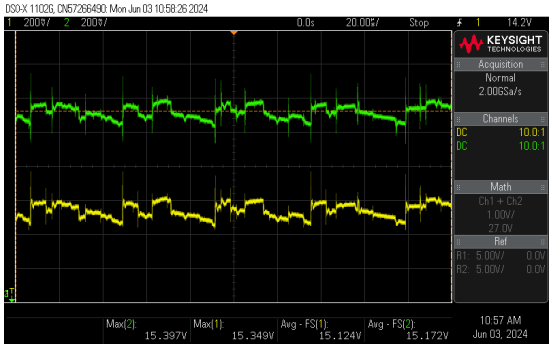


Figure 17: Typical IM+ (Green) and VM (Yellow) Voltages

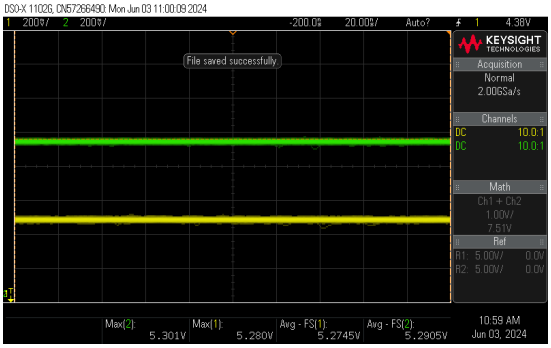


Figure 18: Typical I5+ (Green) and 5V (Yellow) Voltages

When the ESP32 and Raspberry Pi are connected, the potential difference between 5V and I5+ is $> 12mV$. Oscilloscope measurements (Figure 18) show little variation in 5V and I5+.

The 5V pin was measured as constant $\approx 5.3V$, giving:

$$P_{5V} = I_5 \times 5.3 \tag{11}$$

Since V_{BAT} is constantly changing, it must be measured. I_M and I_5 are calculated with equation (10). Methods to detect the required potential differences are discussed later in this section.

5.2.2 Measuring Battery Voltage and Integrating Power Management

There are many methods to step down V_{BAT} so the ESP32 can measure it [12]:

1. Buck SMPS
2. Adjustable linear voltage regulator
3. Potential divider

Analysing these three options, the potential divider is the simplest and most cost-effective approach. However, resistors are prone to noise, which influences measurement accuracy. Therefore, the circuit (Figure 19) includes a potential divider with a buffered RC in a low-pass filter configuration ($f_c = 1.45Hz$). The capacitor to ground is a parallel combination of ceramic and electrolytic capacitors; ceramic capacitors are effective at filtering out high-frequency noise due to their low ESL and ESR, whereas electrolytic can smooth out large voltage fluctuations because of their high capacitance.

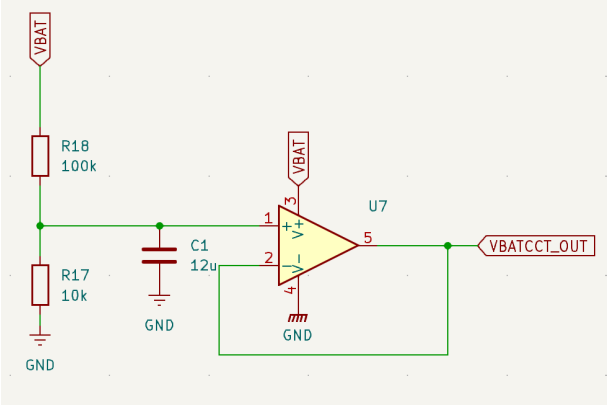


Figure 19: Battery voltage step-down circuit diagram

The relationship between V_{BAT} and V_{out} is used to recover VBAT digitally:

$$V_{out} = 0.0909 \times V_{BAT} \tag{12}$$

Initial tests using the NiMH batteries gave $V_{out} = 1.551V$ with $V_{BAT} = 16.03V$, i.e. a percentage error of $EPE = 6.43\%$, which is acceptable.

5.2.3 Circuits for Measuring Current

Table 6 discusses four circuits that translate IM+, VM, I5+ and 5V into voltages that can be read by the ESP32 to calculate current.

After reviewing each option, the differential amplifier was chosen because of its cost and accurate output.

Method	Description	Advantages	Disadvantages
Voltage Divider	Step down individual voltages and input into ESP32's analogue pins.	• Easy to implement. • Requires less hardware. • Zero cost (components sourced from labs).	• Low resolution due to noise. • Uses 5 pins instead of 3.
Differential Amplifier	Amplify potential difference across shunt resistors and send to ESP32 [13].	• Extremely accurate with right opamp. • High CMRR and complex filters. • Fairly cheap.	• Hard to find suitable opamp. • Input offset may be amplified.
High-Side Current Sensing	Use voltage-to-current opamp circuit where output voltage matches current in 10mΩ.	• Uses less hardware than differential amplifier. • Less noise due to fewer resistors.	• Costlier and more complex than differential amplifier. • BJT may have different operating points. • Current needs amplification.
Premade ICs	Use premade through-hole differential amplifiers or current sensors (e.g., AD629, INA219) [14].	• Least overall hardware. • Very accurate with right chips.	• Many chips don't meet all supply and output requirements. • Very costly. • Often not through-hole. • Internal shunt resistor.

Table 6: Comparison of Voltage Measurement Methods

5.2.4 Choosing an Amplifier

It was established that the opamp must:

1. Have a supply voltage of +18V for a single rail.
2. have CMRR $> 100dB$
3. have input offset voltage of $< 500\mu V$
4. support rail-to-rail operation
5. be low cost ($<$ £5)

After considering various options, it was not possible to find a through-hole opamp that satisfied all the criteria. The OPA2241 precision amplifier was selected as it fulfilled all specifications except complete rail-to-rail input/output (data sheet in Appendix D).

5.2.5 Circuit and Testing

The output voltage of a differential amplifier circuit (Figure 20) is:

$$V_{out} = \frac{R_f}{R_1}(V_2 - V_1), \text{ if } R_f = R_g \ \ \& \ \ R_1 = R_2 \tag{13}$$

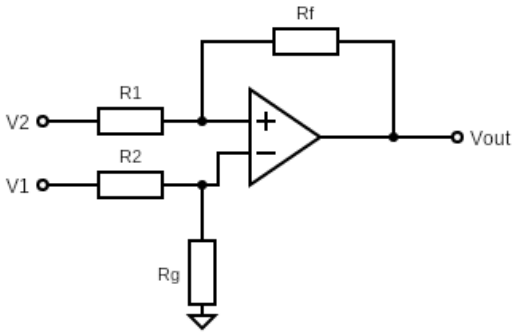


Figure 20: Differential amplifier circuit diagram

The chosen opamp has common-mode input voltage range: $V_- - 0.2$, $V_+ - 0.8$. Therefore, to amplify the difference between IM+ and VM, the individual voltages are halved before being fed as inputs. We use a potential divider consisting of $10k\Omega$ resistors with 1% tolerance.

Differential amplifier gains for the motor and 5V voltages were chosen so that the expected output range covers the $0 - 3.3V$ range. A gain of 200 was chosen for the IM+/VM circuit, and a gain of 330 for the I5/5V circuit. Resistors were then chosen accordingly.

Digital processing of V_{out} is required to recover the true potential difference. From here, currents I_M and I_5 can be calculated using equation 10:

```
const float CURRENT_SENSE_RESISTOR = 0.01;
const float USB_Ratio = 330;
const float IM_Ratio = 200;
const float IM_DIVIDER_RATIO = 0.5;

//Convert raw ADC values to voltages
float i5_voltage = ((i5_raw / (float)ADC_RESOLUTION) * ADC_MAX_VOLTAGE);
float im_voltage = ((im_raw / (float)ADC_RESOLUTION) * ADC_MAX_VOLTAGE);

float current_5v = i5_voltage / (CURRENT_SENSE_RESISTOR * USB_Ratio);
float current_motor = (im_voltage/IM_DIVIDER_RATIO) /
                      (CURRENT_SENSE_RESISTOR * IM_Ratio);
```

Finally, power consumption can be calculated:

```
float power_consumption_5v = 5.3 * current_5v;
float power_consumption_motor = battery_voltage * current_motor;
total_power_consumption = power_consumption_5v + power_consumption_motor;

// Calculate battery energy in Watt Hours
float used_capacity_mAh = (total_power_consumption * elapsed_time_hours * 1000)
                / battery_voltage;
```

Voltage follower circuits are placed between inputs and amplifier stages to eliminate interference. Thus, the circuit diagram is shown in Figure 21. The physical implementation is shown in Figure 22.

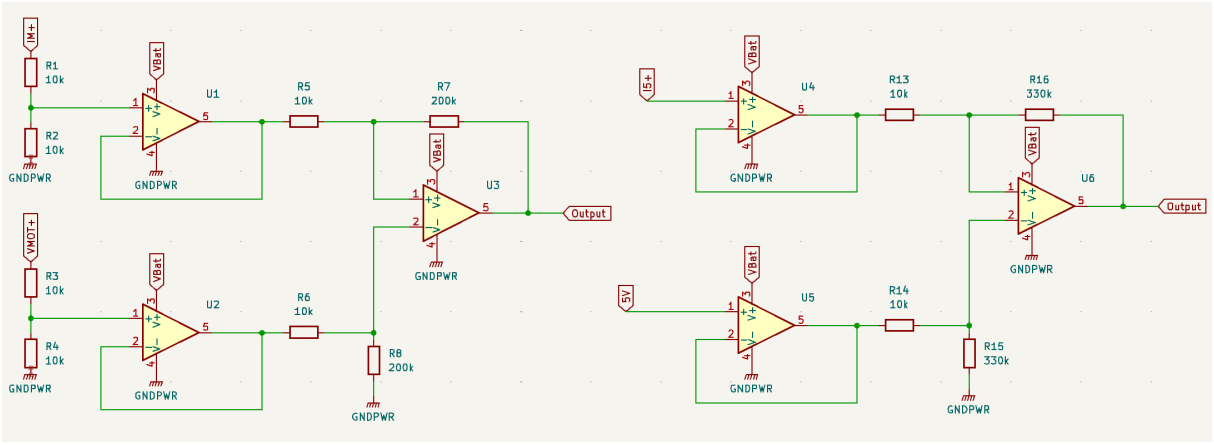


Figure 21: Differential Amplifier Circuits for IM+/VM and I5/5V measure

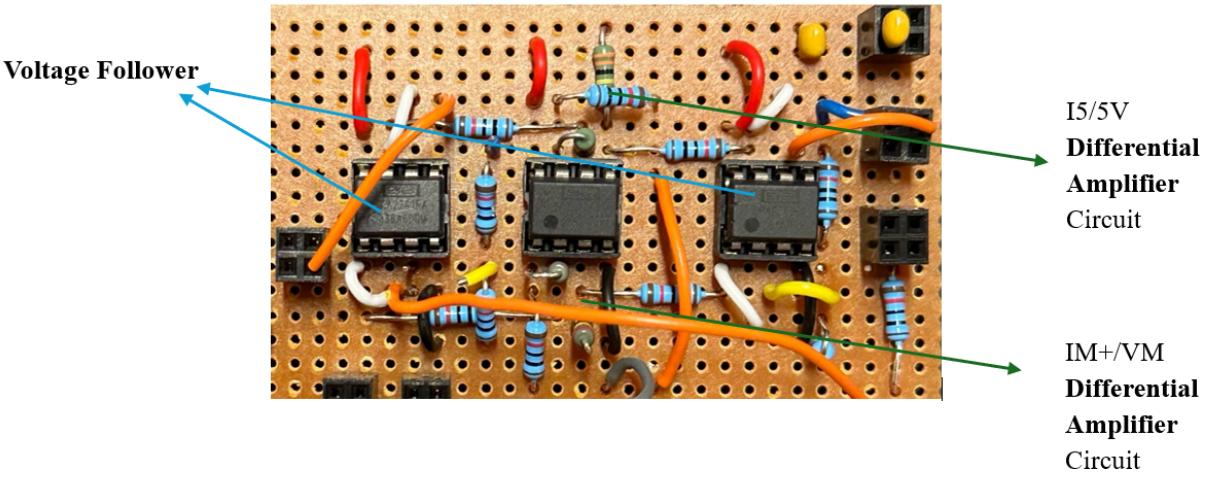


Figure 22: Physical circuit implementation

5.2.6 Circuit Testing

The potential difference between IM+ and VM was recorded at $14mV$, and the output of the differential amplifier ($Av = 200$) is $2.8866V$. Hence, there is a percentage error of $EPE = \frac{(14\times0.2)-2.8866}{14\times0.2} \times 100 = 3.09\%$. The recorded difference between I5 and 5V was $6.3mV$, and the output of the differential amplifier ($Av = 330$) was $2.1877V$.

Hence, the percentage error is $EPE = \frac{(6.3\times0.33)-2.1877}{6.3\times0.33} \times 100 = 5.23\%$. This concludes that both outputs from the amplifier are accurate.

5.2.7 Pin Protection

The operational amplifier rails are kept at battery voltage ($\approx 16V$), yet the ESP32 analogue pins only tolerate voltages slightly above $3.3V$. Therefore, a final circuit must include some form of pin protection that ensures the output voltage from the differential amplifiers never exceeds this limit [15].

A simple circuit comprising a $3.3V$ rail, two zener diodes, and a resistor can be used (Figure 23). If the input voltage exceeds the rail by $0.2V$, the diode will conduct, resulting in the ESP32 analogue pin only receiving $3.5V$. The physical implementation is shown in Figure 24.

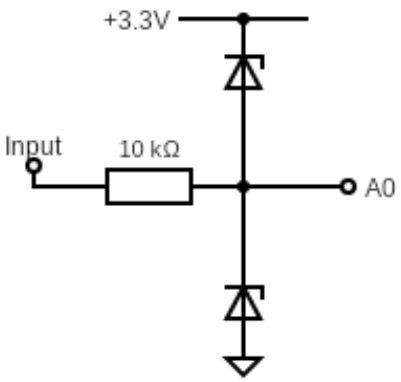


Figure 23: Pin protection schematic

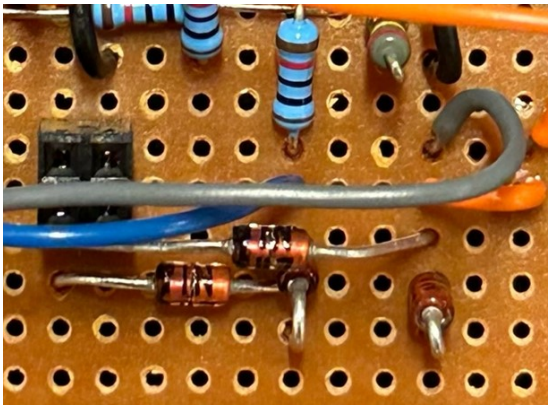


Figure 24: Physical pin protection implementation

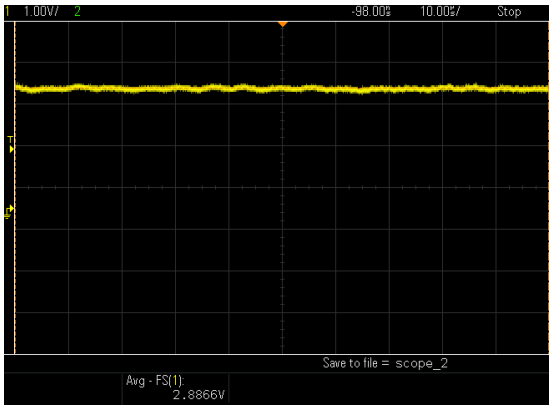


Figure 25: Stepped-up potential difference between IM+ and VM

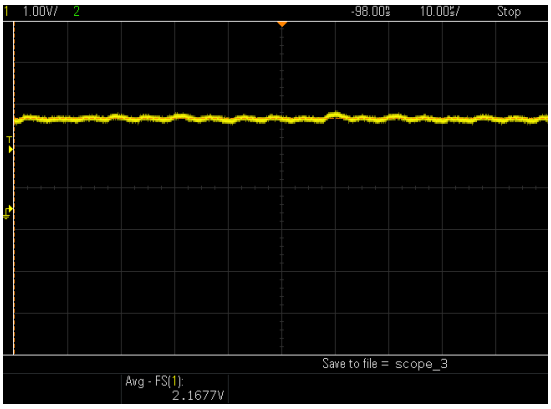


Figure 26: Stepped-up potential difference between I5+ and 5V

Repeating the testing outlined in the previous section gave different output voltage readings. This is due to the forward potential drop across the Zener diode. The results only differed by ±10-30mV, slightly increasing the experimental percentage error.

5.3 Battery Capacity

5.3.1 Charge Counting vs. Look-up Table

Two methods of determining battery capacity were considered [16]:

1. Charge counting
    - The process starts with an initial state of charge (SoC).
    - The current is integrated to calculate the total charge removed.
    - SoC is updated.
2. Voltage look-up table
    - Voltage against graph SoC needs to be characterised.
    - The graph is implemented as a look-up table.

While charge counting is the industry standard, it requires extremely accurate information about total battery charge. This is difficult since the project involves changing the batteries often. While our current

measurements are sufficiently accurate, integration over time will lead to an inevitable accumulation of errors. Using a voltage look-up table means that each measurement is independent.

5.3.2 Characterising the Batteries

The batteries must be characterised to implement a look-up table. Multiple batteries were discharged at a current of 1.15A (the typical current draw while balancing). The battery voltage and current were sampled every $0.5s$ to produce graphs [17] of battery voltage against charge out (Figure 27).

The collected data was averaged and doubled to give an average discharge curve for two batteries in series. Although the averaging process loses some accuracy, we prioritised the metric's reliability.

$16.19V$ was chosen as the 100% battery point, and $13.5V$ for 0%. A voltage is mapped to a battery percentage according to charge out. Figure 28 shows the finalised function. This graph is implemented as a lookup table using `std::unordered_map`, resulting in a constant lookup time.

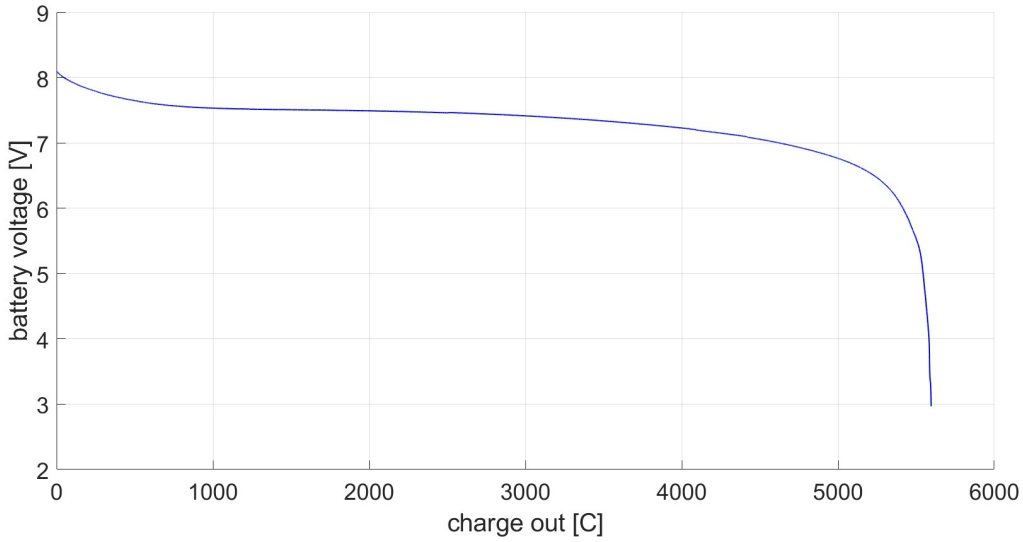


Figure 27: Measured discharge curve example (Battery #2)

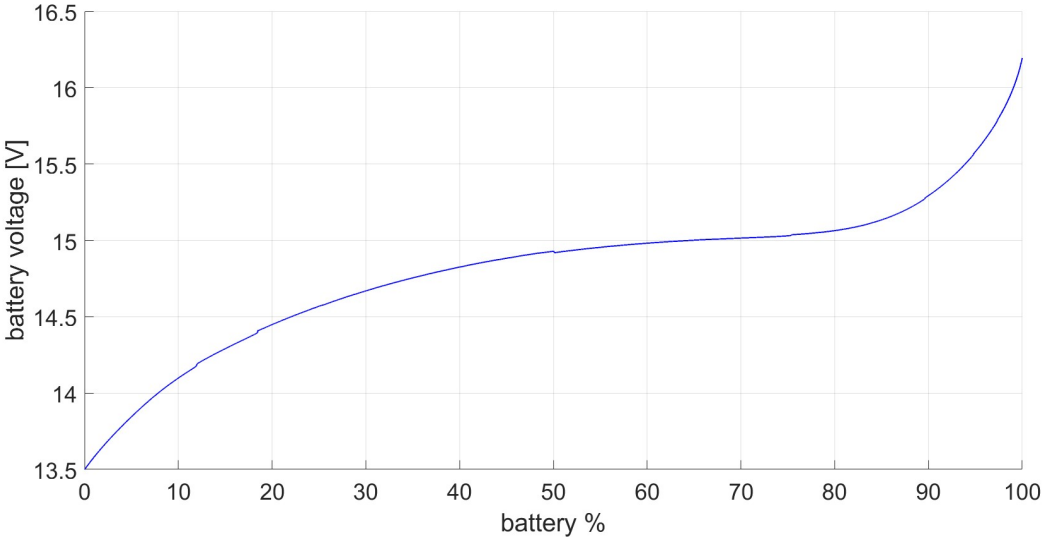


Figure 28: Graph of battery voltage against battery percentage

5.4 Integrated Subsystem

We uploaded the digital processing code to the ESP32 to test the overall subsystem and connected the final circuit (Figure 29) to the robot while balancing.

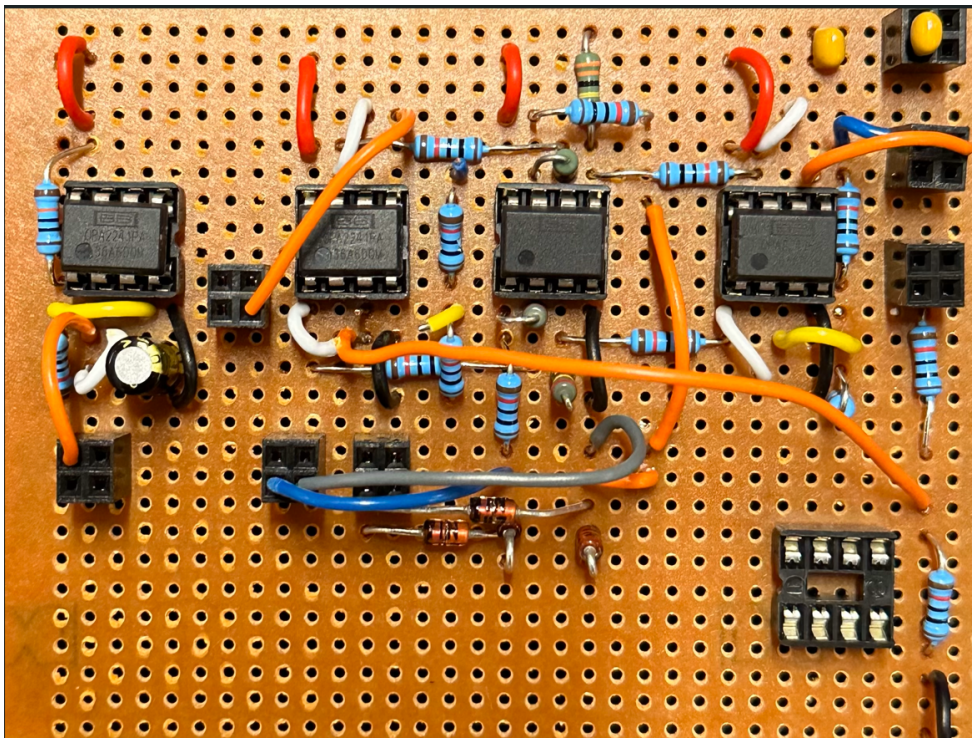


Figure 29: Fully implemented power module on stripboard

The outputs from the serial monitor were consistent with what was expected:

- VBAT = 15.86
- Motor Current = 0.32A
- Serial Port Current = 0.12A

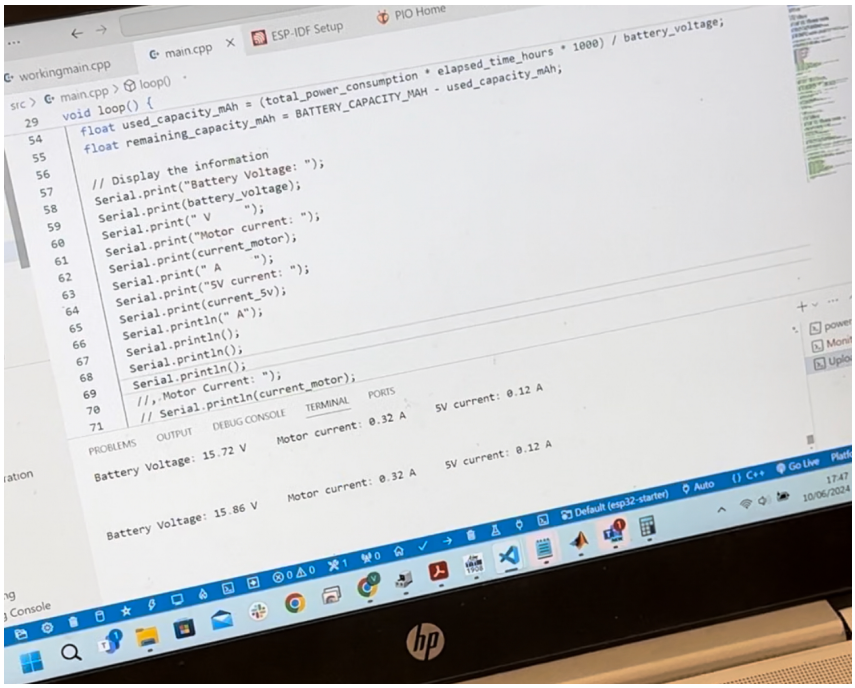


Figure 30: Image of Serial Monitor

5.5 Evaluation and Improvements

5.5.1 Evaluation of Testing Process

All circuits were first designed on KICAD and simulated on LTSpice (Figure 31). This helped model the theory but excluded practical considerations such as noise and breadboard/wire reactance.

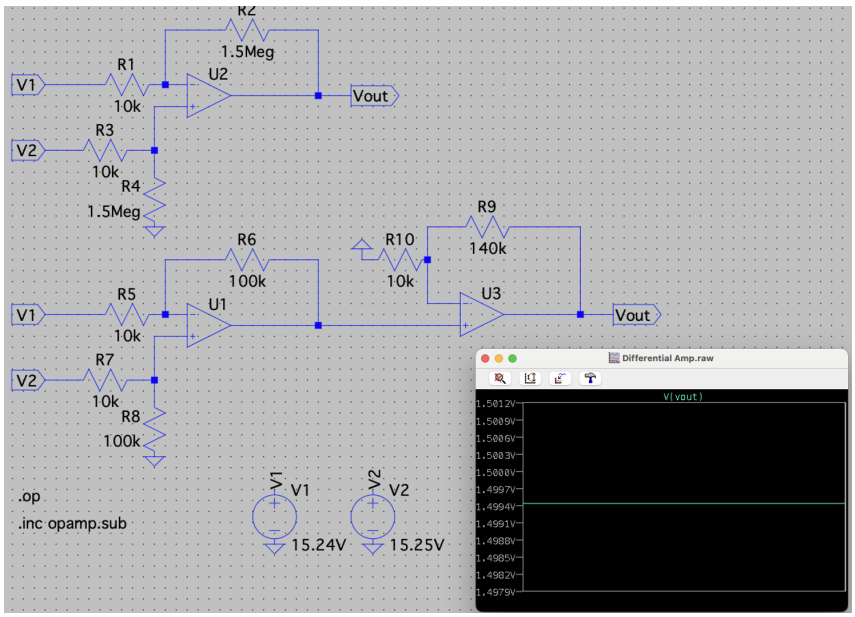


Figure 31: LTSpice circuit simulation

Final subsystem testing was involved before being soldered onto the stripboard. The circuit was tested using the actual robot in static and dynamic conditions. Refinements were constantly made according to the agile sprint methodology, e.g. including capacitors to filter noise on power rails and outputs or devising the pin protection circuits.

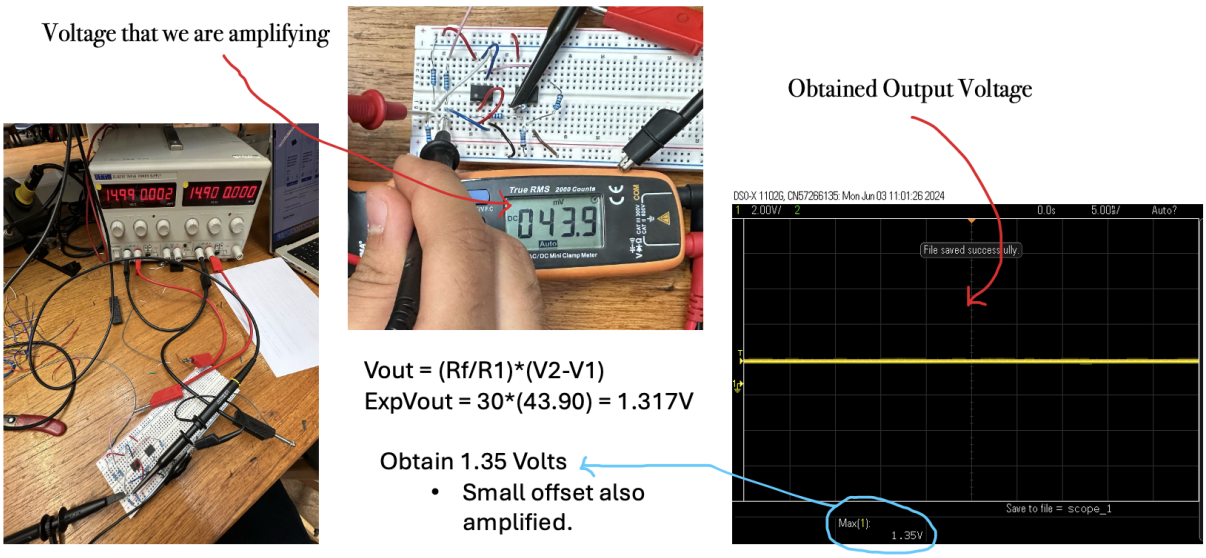
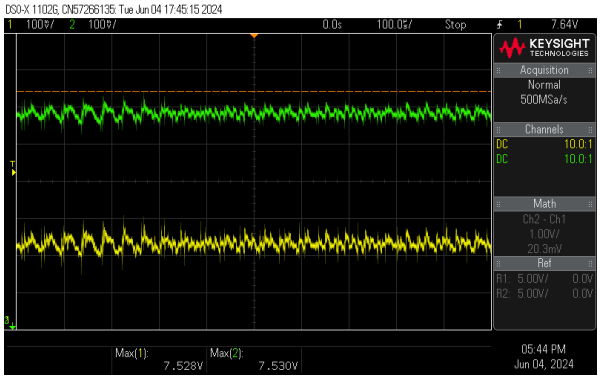
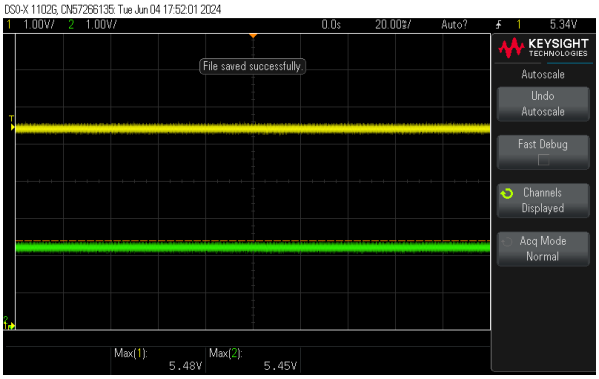


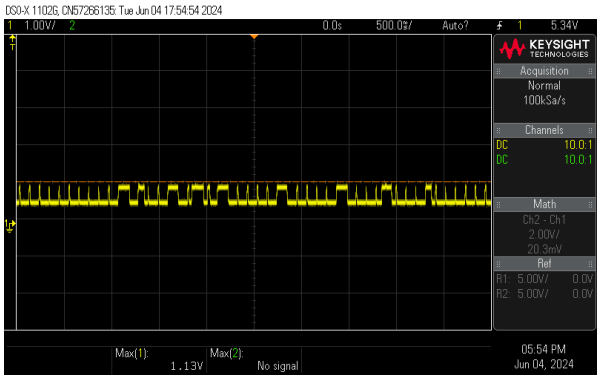
Figure 32: Testing with PSU



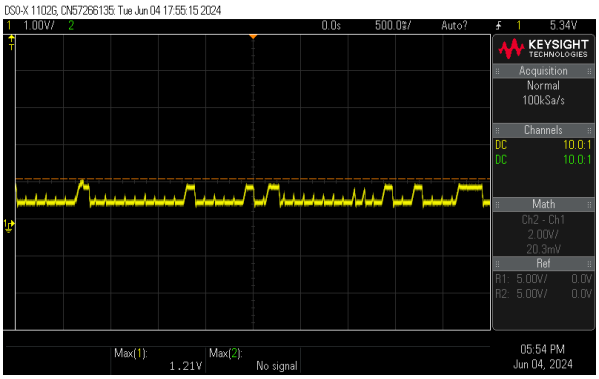
(a) Stepped Down IM+ and VMOT when Balancing



(b) I5 and 5V in Static Condition



(a) I5 V5 Output Voltage without Rail Filtering



(b) I5 V5 Output Voltage With Rail Filtering

Figure 34: Oscilloscopes Readings

The overall testing process can be improved by conducting more trials to identify anomalies and recording how uncertainties are introduced in the system due to the accuracy of each measuring device.

5.5.2 Evaluation of Measurement

The current battery management module has a few shortcomings. Firstly, all circuits have a non-negligible degree of error. This error can be attributed to external noise, component overheating, or resistor tolerances. Future work would involve research into more complex circuitry, like a high-order Butterworth filter and high-accuracy components.

Secondly, battery characterisation was conducted under constant current, a range of loads should be considered in future work. Moreover, the process could be more precise by thoroughly investigating relevant electrochemical reactions, as in industry. Finally, the battery look-up table currently only contains values to two significant figures. This can be increased once a more accurate circuit is in place.

Although the OPA2241PA opamp has a CMMR of $124dB$, it has an input offset voltage, which, when amplified with a gain of 200/330, offsets measurements by $10mV/16.5mV$. This effect is fairly small compared to the actual measurements. To combat this, future research should be conducted into the OPA2188 or the OPA197, which boast lower input offset voltages and rail-to-rail input/output despite their scarcity and higher cost.

Nevertheless, the power management module exhibits high accuracy of $100 - MAX\_ERROR = 94.77\%$ with a cost-effective and simple design.

6 Autonomous Mapping and Exploration

6.1 Brief

This section will discuss the techniques for autonomous navigation inside an unknown environment: the Simultaneous Localisation and Mapping (SLAM) algorithm for mapping, and the Frontier-based approach for autonomous exploration.

6.2 SLAM

SLAM is a robotic mapping technique used by autonomous robots to build a map of an unknown environment, whilst simultaneously keeping track of their current location within the map. The robot does so by tracking distinctive features in the environment and estimating its position relative to the features.

SLAM provides two key advantages:

1. SLAM does not rely on GPS and can work in GPS-limited indoor areas
2. SLAM can respond quickly to a dynamically changing environment

The two most researched SLAM techniques are Visual SLAM (camera-based) and Time-of-flight SLAM (ToF sensor-based).

Initial testing revolved around Visual SLAM, where we tested a monocular camera solution (Logitech C270). A configuration (`.yaml`) file was first created through checkerboard-based camera calibration, then the real-time SLAM library 'ORB-SLAM' was run. However, the instability and constant resetting of mapping attempts plagued the setup.

We considered usage of RGB-D (depth) and stereo cameras, which have depth-sensing capabilities. This was overruled due to their expensive pricing. There were cheaper but bulkier options, which would adversely affect the balancing of the robot.

6.3 LiDAR

Our design uses the direct time-of-flight LD06 LiDAR (Figure 35), a very low-cost mini-sized LiDAR with a 12-metre scanning radius and 4500 Hz sampling frequency [18].

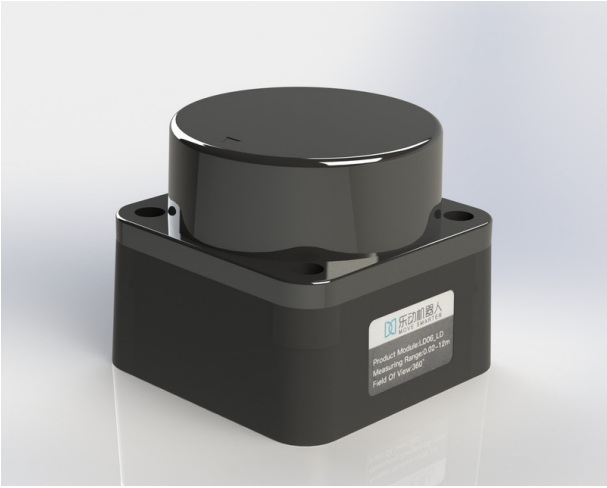


Figure 35: LD06 LiDAR

LiDAR operates as a "time-of-flight" sensor. It emits a modulated laser pulse which is reflected off an object, and measures the time taken for the laser pulse to travel to the object and back. The points measured can be mapped to a 2D coordinate map by considering: (1) time of measurement, (2) distance measured, (3) angle of rotation.

Implementation of this LiDAR informed several considerations:

(a) Head unit design

The head unit was designed to be 3D-printed and screw-mounted, with careful consideration in avoiding obstructions that disrupt the LiDAR scan.

(b) Tilt angle range

As we used a 2D LiDAR, it had limited vertical angle resolution for scanning. We initially considered the trigonometry of LiDAR scans when the bot was tilted, but testing showed limited effect on the LiDAR data.

(c) Demonstration environment

The arena walls of the demo environment has to be at least as tall as the LiDAR's height on top of the robot, such that the LiDAR can operate within a enclosed environment, mimicking an actual warehouse.

6.4 PiCamera and Incident Detection

The Raspberry Pi Camera Module 2 was used. It includes a 8MP Sony IMX219 sensor, which provides a 1080p video streaming resolution, sufficient for incident detection.

In our demo environment, incidents are represented by green-coloured crates. We utilize the camera module and the OpenCV library to perform color thresholding on a Raspberry Pi. Figure 36 shows the OpenCV colour detection results.

When a green crate is detected, indicating an incident, the Raspberry Pi sends a copy of the current map and images of the incident to the website server. The website server receives these files and updates the user interface (UI) to display the incident details. For more information on the UI, refer to the section of the report.

(a) Crate not detected

(b) Crate detected

Figure 36: OpenCV colour detection

6.5 Robot Operating System 2

Robot Operating System 2 (ROS 2) was used to connect the implemented nodes. In ROS, nodes share information by creating a publisher object for a specific topic, which other nodes subscribe to to receive information. This decoupled communication allows nodes to interact without needing to know about each other's existence, allowing for a modular and flexible system design.

6.6 Nodes of the system

To achieve our application, we created several local processing ROS nodes. This subsection discusses the major processing nodes used.

6.6.1 Cartographer Mapping

Cartographer was used to implement SLAM. It uses LiDAR scan data and odometry data to build a map of the surroundings.

Cartographer operates on two subsystems - Local and Global SLAM.

Local SLAM's job is to build a series of submaps. Figure 37 details the process:

1. Extrapolation

   Based on the most recent known pose (position and orientation) and the sensor data, the extrapolator predicts the robot's pose at the time of the new scan.

2. Scan matching

   The predicted pose serves as the initial guess for the scan matching process. This makes the scan matching more efficient because the algorithm starts with a close approximation of the correct pose.

3. Pose correction

   During scan matching, the algorithm adjusts the initial guess to achieve the best fit between the new scan and the existing submap. This optimised pose is then used to update the submap and the robot's pose estimate.

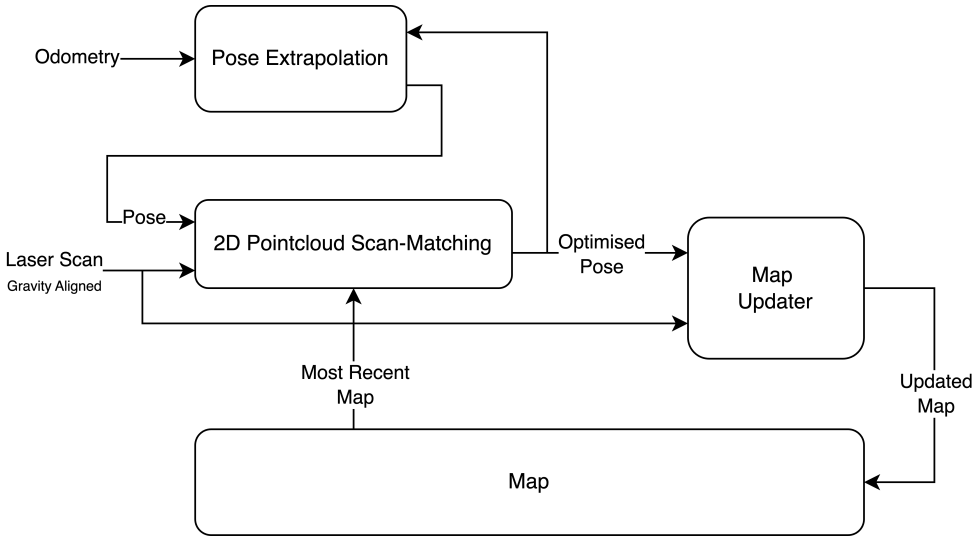


Figure 37: Local SLAM

While the local SLAM generates its succession of submaps, Global SLAM runs a global optimization task runs in background. Its role is to re-arrange submaps between each other so that they form a coherent global map [19].

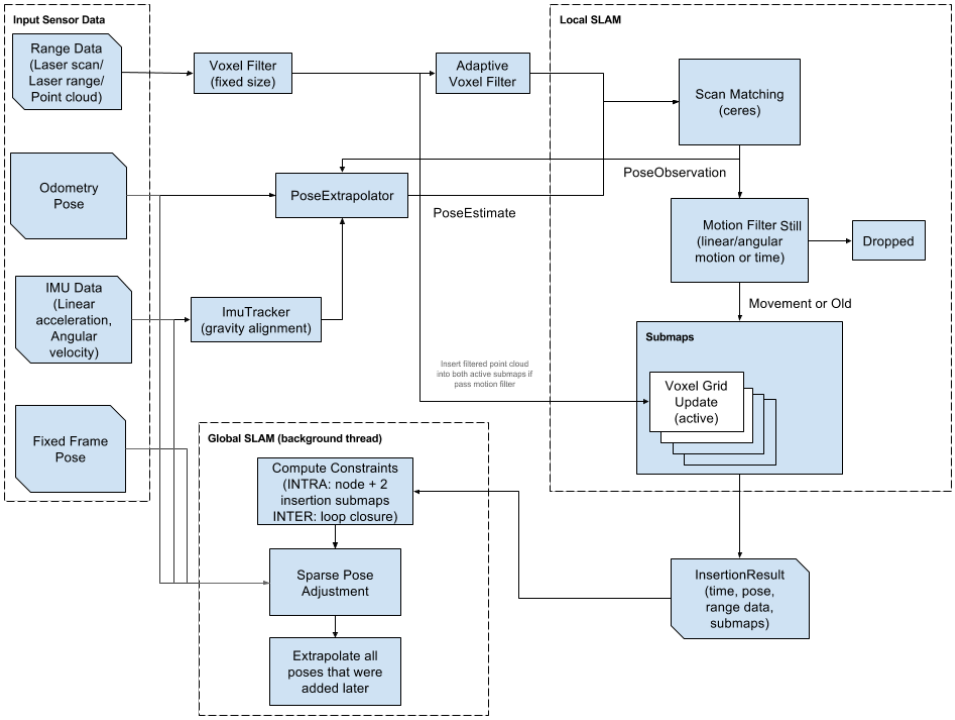


Figure 38: Cartographer Diagram (Local and Global SLAM) [19]

Cartographer's global SLAM is a type of "GraphSLAM"[19], where the problem is represented as a "pose graph". Nodes in this graph represent robot poses, and edges represent spatial constraints between these poses. Its goal is to find the best configuration of robot poses that minimizes the error between predicted and observed sensor data.

Such optimisation of constraints can be visualised as tightening small "rubber bands" that connect all the nodes in Figure 39 below:

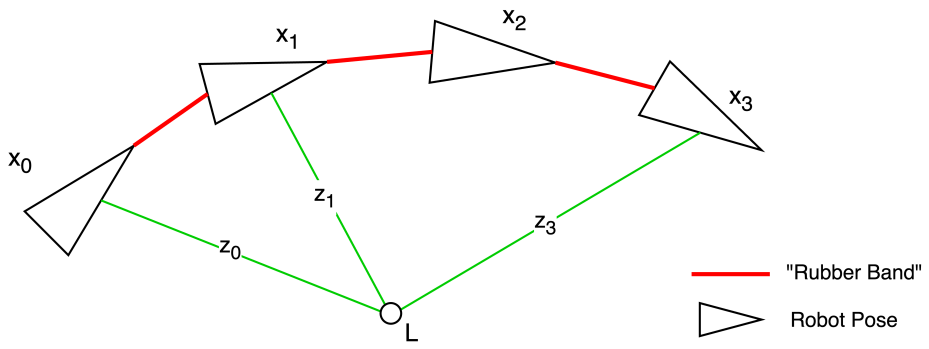


Figure 39: Pose error estimation

6.6.2 Autonomous Exploration

SLAM helps map and track the robot's position, but the robot requires an algorithm to decide where and how to autonomously explore an unknown environment.

There are no ROS 2 libraries which implement autonomous exploration. Hence, we created a ROS package using a Python script which implements autonomous exploration (see Appendix A).

Functions include:

(a) Frontier exploration

There are several unexplored edges in the environment called "frontiers" which the robot could explore next (Figure 40). Larger frontiers reveal larger unexplored areas, and closer frontiers are easier to get to. The frontier exploration algorithm balances both size and distance to pick the next frontier to explore.

Firstly, the algorithm identifies "frontier points" by looking for cells in an occupancy grid that are free and adjacent to unknown cells. Then, it uses a depth-fist search to group connected frontier points. Lastly, it calculates a score for each "frontier group" by dividing the group's size by its distance from the robot. The group with the highest score is selected as the target.

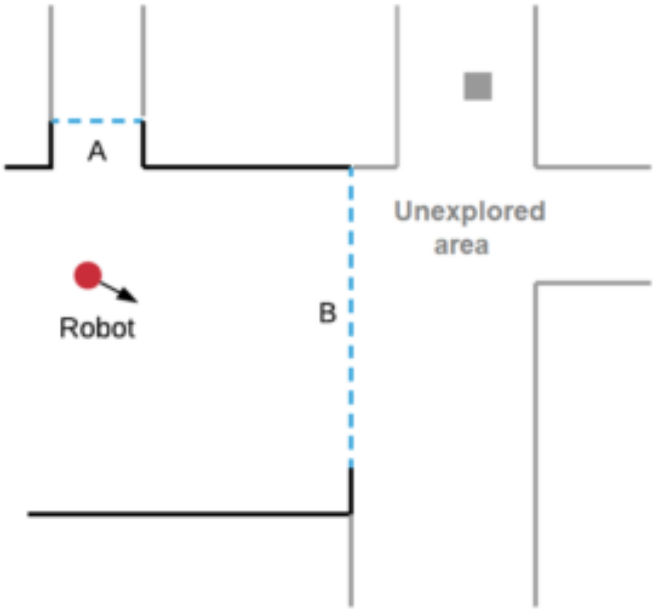


Figure 40: Frontiers groups of the mapping environment [20]

(b) Local path-finding

Once the robot has decided on exploring a frontier, it must create a path from its current location to the frontier. The A* algorithm is used to find a minimum path from the robot's current position to the centroid of the selected frontier group. The path is then smoothed using B-spline interpolation to generate a smooth trajectory for the robot to follow.

(c) Pure pursuit

Pure pursuit computes the necessary linear and angular velocities to keep the robot on the calculated path. This involves picking a point on the path a certain "lookahead distance" ahead of the robot's current position and making the robot drive towards it, as in Figure 41.

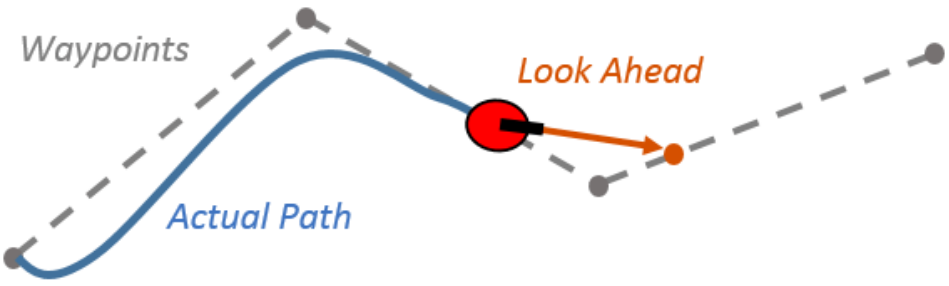


Figure 41: Pure pursuit [21]

6.6.3 Other nodes

Other nodes include:

(a) rf2o laser odometry

Uses LiDAR scan data to produce odometry data (estimation of robot's positional change over time).

(b) ldlidar SDK

SDK node for transforming LiDAR raw scan data into 2D pointcloud.

6.7 Testing and Tuning

Several improvements were made to smoothen the navigation process:

(a) Cartographer SLAM tuning

Manual testing showed that the mapping was really sensitive to small jerky rotations.

The motion filter of Cartographer SLAM was tuned to filter these small jerky motions. The filter removes scan data-points if the motion that led to it is lower than a certain threshold.

By increasing the angle threshold, Cartographer was less sensitive to small movements, stabilising the map creation. Figure 42 shows successful map building of our built arena during top-level testing after SLAM tuning.

(b) Pathfinding cost-map

Testing showed that the robot was moving too close to walls and periodically crashing. We included a cost-map (Figure 43), which assigns each cell a cost based on its proximity to walls. This allows us to prioritise paths with lower costs, keeping the robot away from walls.

(c) Pure pursuit look-ahead distance

Tuning the look-ahead distance was crucial - too far, and the robot cuts corner; too close, and the robot over-corrects and oscillates (Figure 44).

Extensive testing was done to find an optimal look-ahead distance which balances these issues.

(d) Pure pursuit velocity commands

We noticed significant map drift when the robot exhibited turning motion (simultaneous linear and angular motion). To reduce map drift, we modified the pure pursuit function to ensure that only one type of motion (linear or angular) is commanded at a time. This also ensures better stability of the robot's balance.

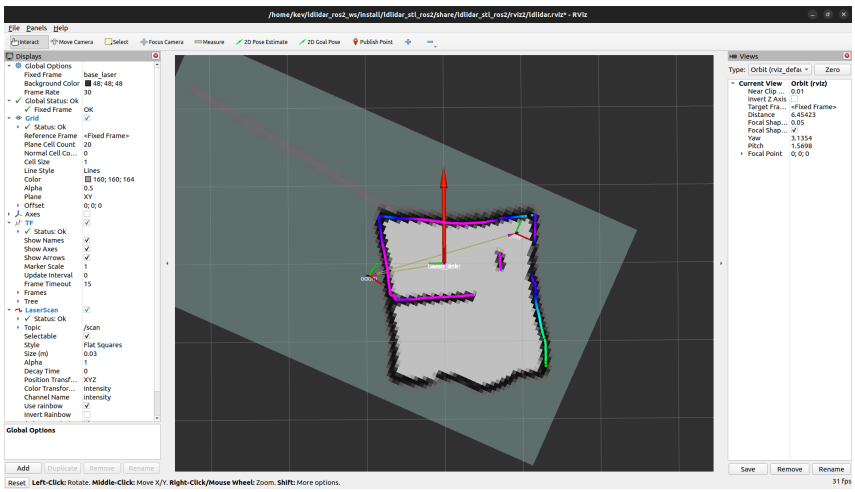


Figure 42: SLAM mapping during top-level testing

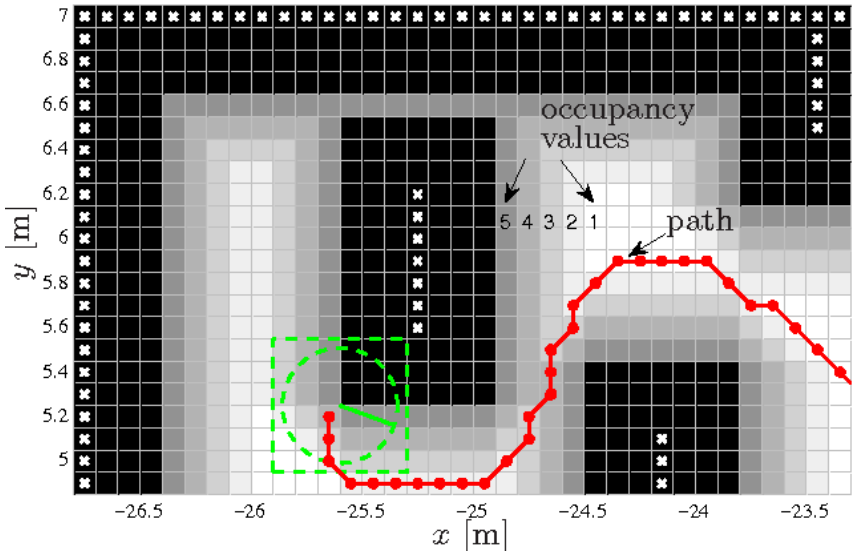


Figure 43: Visualisation of 2D Costmap [22]

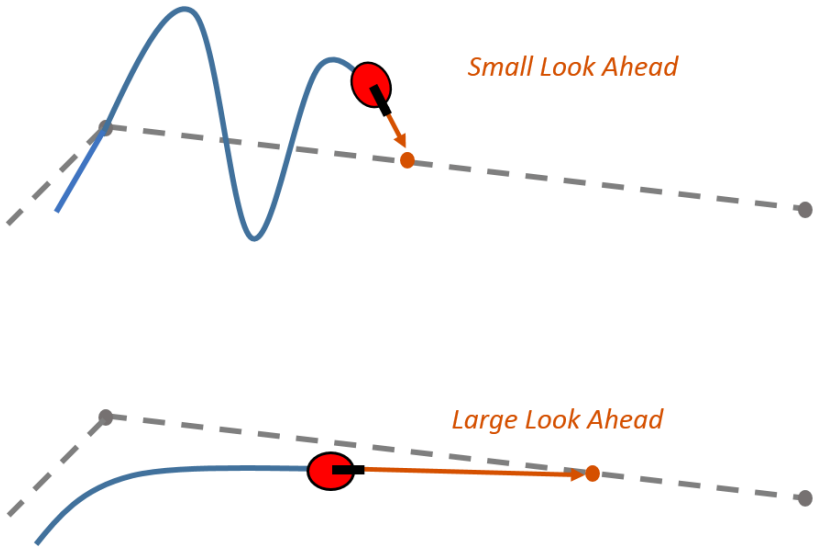


Figure 44: Pursuit with lookahead [21]

6.8 Failure Mode and Effect Analysis

One major failure mode considered is when the autonomously navigating bot topples over and requires rescue. Our solution was to send a copy of the current map and and current position of the bot to the website, and indicate request for rescue. More information about this can be found in Appendix C.

7 Software System

7.1 User Requirements

The group agreed on an initial set of requirements for the overall software systems. The user should be presented with:

1. a dynamically updating feed.
2. a joystick to control the robot.
3. an icon for battery usage, like a phone.
4. an interface to send extra commands to the robot.
5. a map on display given by our application team.
6. a dashboard view to monitor historical data.
7. a camera stream.
8. an aesthetically pleasing interface.

Some requirements, such as the joystick and live camera stream, were later dropped as the project focused on its core features.

Overview System Architecture

Communication between that ESP32 and a client can be configured via WiFi. Figure 45 shows an overview of the architecture used to establish this communication. The architecture will be elaborated on in the following sections.

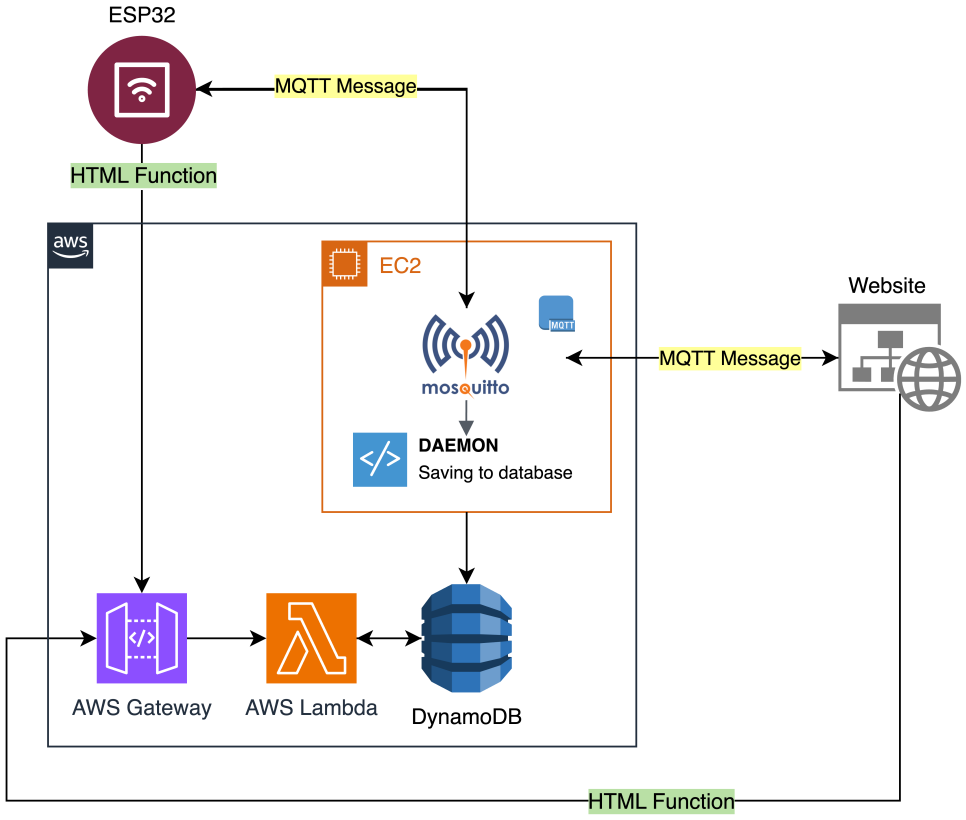


Figure 45: Overall Architecture for Interactions between the ESP32 and the Website

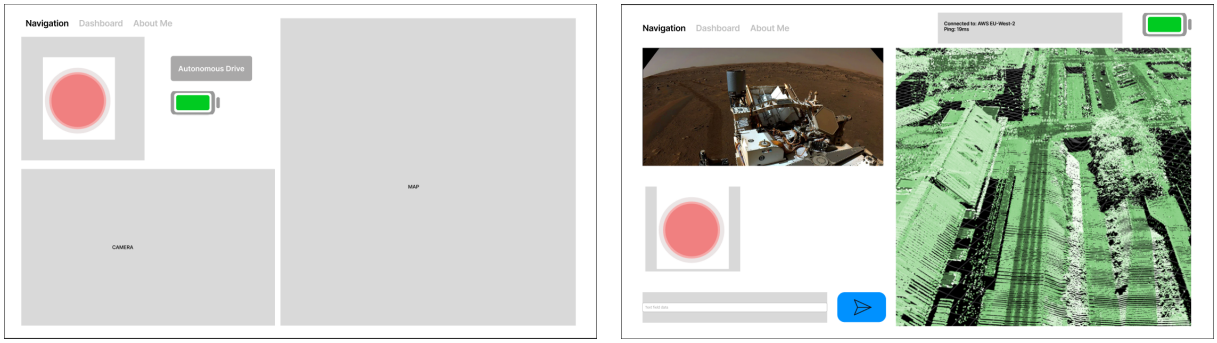
7.2 Front-End

7.2.1 Rapid Prototyping

Based on the above requirements, a sitemap was generated, simplifying of all of the features, and modelling the user flow:

Figure 46: Sitemap showing user flow between landing/control/debug pages

Initial drafts were developed on an online tool, Figma, allowing designs to be quickly revised without requiring extensive code changes. Figure 47 shows an early iteration illustrating considerations to avoid blocking the camera stream with the thumb.



(a) First iteration

(b) Second iteration

Figure 47: Rapid iterations of the design updated to reflect UX features

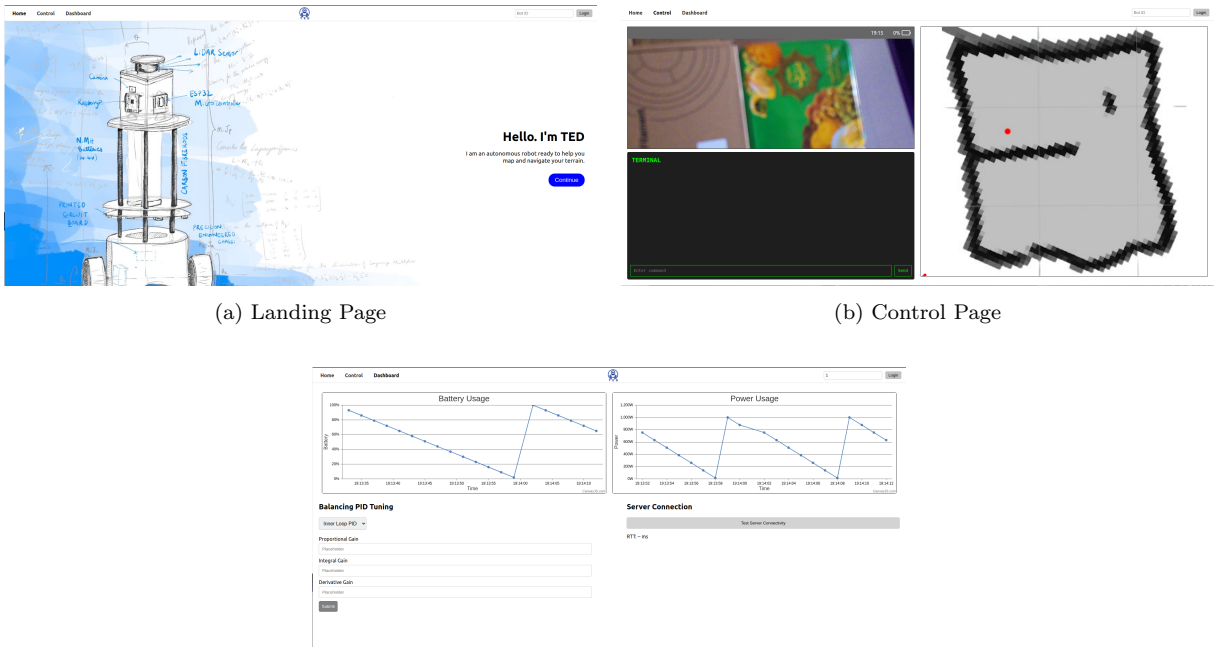
Although this tool can also be used to generate source code, this feature was not used as it introduced a lot of undesired elements such as extra frames that would ultimately need to be manually resolved.

7.2.2 Front-End Implementation

The group chose to use the React framework for the final implementation as it is:

- Dynamic, meeting the requirements outlined above.
- Widely adopted, meaning robust support.
- Flexible and modular in design.

Concept art was created by hand using digital design tools on a graphics design tablet (Procreate), influenced by the English tradition of watercolors [23]. The results can be seen in the landing page (Figure 48a) along with the rest of the front-end elements in Figure 48.



(a) Landing Page

(b) Control Page

(c) Dashboard Page

Figure 48: UI Showcase

7.3 IoT Device Considerations

7.3.1 Choosing an Application Layer Protocol

This section addresses how data can be exchanged between the user and the robot to provide real-time updates, specifying the protocols and the procedure for its implementation. The following constraints had to be considered:

1. low resource-usage: the ESP32 is a constrained device with only 520KB of RAM.
2. low latency: the client requires real-time control of the robot

Multiple application layer protocols were considered; a selection of them can be found in table 7.

	WebSocket	**MQTT**	**CoAP**	**Zigbee**
Communication Type	Full-duplex, bidirectional	Publish Subscribe	Request Response	Request Response
Connection	Persistent	Persistent	UDP-based	TCP-based
Latency	Low	Low	Low	Low
Efficiency	Moderate	High	High	Medium
Security	SSL/TLS	SSL/TLS	DTLS	DTLS
Resource Usage	Moderate	Low	Low	Medium-High
Requirement	Optional	Broker	Server	Peer-to-Peer

Table 7: Comparing different Application Layer protocols [24] [25] [26] [27]

MQTT was chosen for its reliability, power-effectiveness and low-latency.

7.3.2 Implementing MQTT

MQTT's publish-subscribe protocol efficiently controls the flow of information between the senders (publishers) and the receivers (subscribers) [28].

This architecture is usually best depicted as a client-server, since it requires a third-party broker as an intermediary. The broker distributes messages it encounters, ensuring a node only receives messages pushed to topics that it is subscribed to. This flow is illustrated in Figure 49.

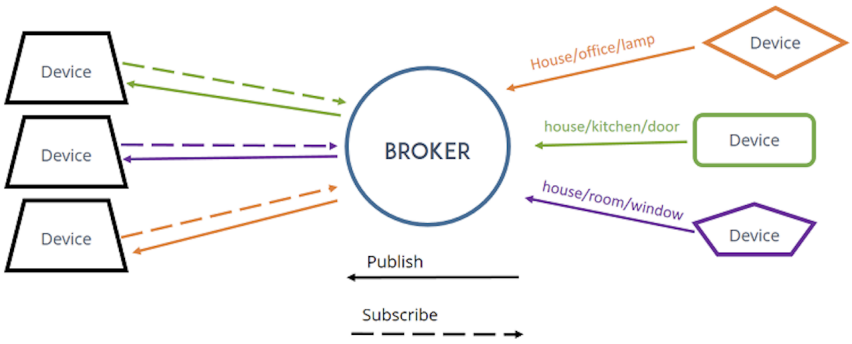


Figure 49: MQTT's Publish Subscribe Protocol [29]

A broker then had to be implemented to establish MQTT connections. Two options were considered:

1. AWS IoT Core [30].
2. EC2 instance running Mosquitto [31].

AWS IoT Core looked like a promising solution due to its seamless interfacing with other AWS services, namely DynamoDB. However, to meet budget constraints, we opted for the Free-Tier EC2 t2.micro instance; IoT Core would have cost £257 [32].

7.3.3 Configuring MQTT

Communications had to be established with the broker. The constrained ESP32 requires a low-overhead link sustained over pure TCP. On the other hand, for web-application, Websockets are better suited. This disposition can be visualised in Figure 50.

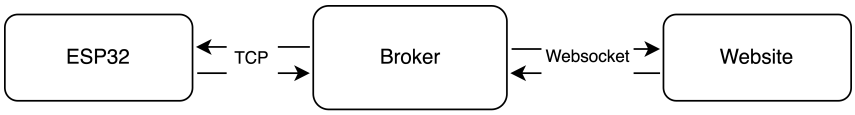


Figure 50: Communication Protocols to connect to broker

Mosquitto then had to be configured to:

1. Accept incoming traffic on pure TCP on port 1883.
2. Accept incoming traffic on WebSockets on port 8000.
3. Secure the MQTT connection by requiring authentication.

A snippet of the configuration file can be found in listing 8. [33]

```
# Default listener for MQTT
listener 1883
protocol mqtt

# Default listener for WebSockets
listener 8000
protocol websockets

# Authentication
allow_anonymous false
password_file /etc/mosquitto/passwd
```

Listing 8: MQTT Configuration File

Number of threads	1000
Ramp-up period	10s
Loop Count	100

Table 8: MQTT Testing Configuration

7.3.4 Testing MQTT

To comprehensively test MQTT, the following framework was adopted:

1. **Connectivity Testing:** Verifying connections can be established with broker and maintained over time.
2. **Performance Testing:** Measuring the latency (RTT) of messages.
3. **Stress Testing:** Simulating a high number of clients to observer system behaviour under load.
4. **Message Flow Testing:** Ensuring messages are correctly routed and received by appropriate subscribers.

This approach was made possible by using Apache JMeter's MQTT plug-in. The following configuration in table 8 was adopted. [34]

Results obtained through data-collection have been aggregated in Figure 51.

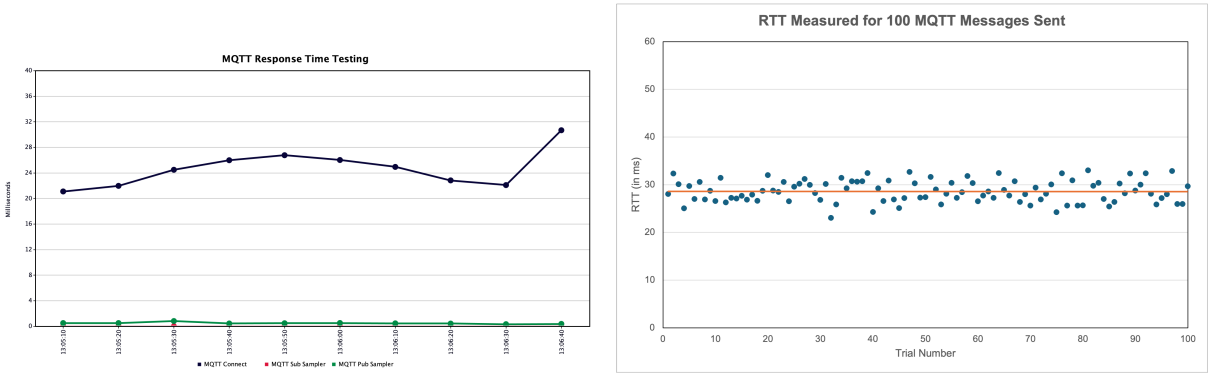


Figure 51: Data collected from testing MQTT with JMeter

The following conclusions were drawn:

- Connection times increased as more users connected to broker. However, a larger instance would address this if project scaled.
- Publishing and subscribing times remained consistently low, indicating effective message handling.
- RTT averaged at 28.7ms, highlighting good MQTT performance.

The MQTT system is subsequently robust for our expected operational load.

7.4 Protocol Design

7.4.1 Exchanging Data via MQTT

To ensure efficient and reliable communications, it is essential to define the:

- type and format of the messages exchanged.
- circumstances of a message being sent.
- actions undertaken when a message is received.
- topics on which messages are sent.
- nodes subscribing/publishing to a certain topic.

Figure 52 presents both the type of the messages exchanged and the interactions data-flow between the ESP32 and the website.

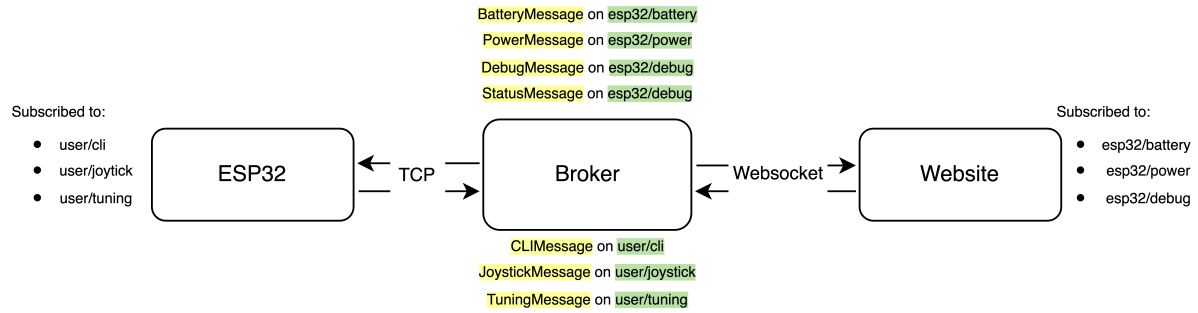


Figure 52: Overview of MQTT Setup

Messages were formatted using the JSON standard for its easy-to-parse nature. *BatteryMessage* was – for example – structured as in listing 9.

```
{
    "runId": INT,
    "timestamp" : INT,
    "battery" : INT
}
```

Listing 9: BatteryMessage Format

The inclusion of *RunId* in the MQTT message helps prevent interference between different devices sharing the same MQTT topic. Indeed, RunId uniquely identifies each device when it is turned on. A full explanation of the process through which RunID is assigned can be found in appendix E.

7.4.2 Publishing Messages

A message should only be published on certain triggers. These triggers have been grouped in Figure 53.

Message	Publish Trigger
Joystick	Key change
Tuning	Submit clicked

(a) Triggers for Website

Message	Publish Trigger
Battery	Every 10 seconds
Power	Every second

(b) Triggers for ESP32

Figure 53: Message Triggers for Different MQTT Messages

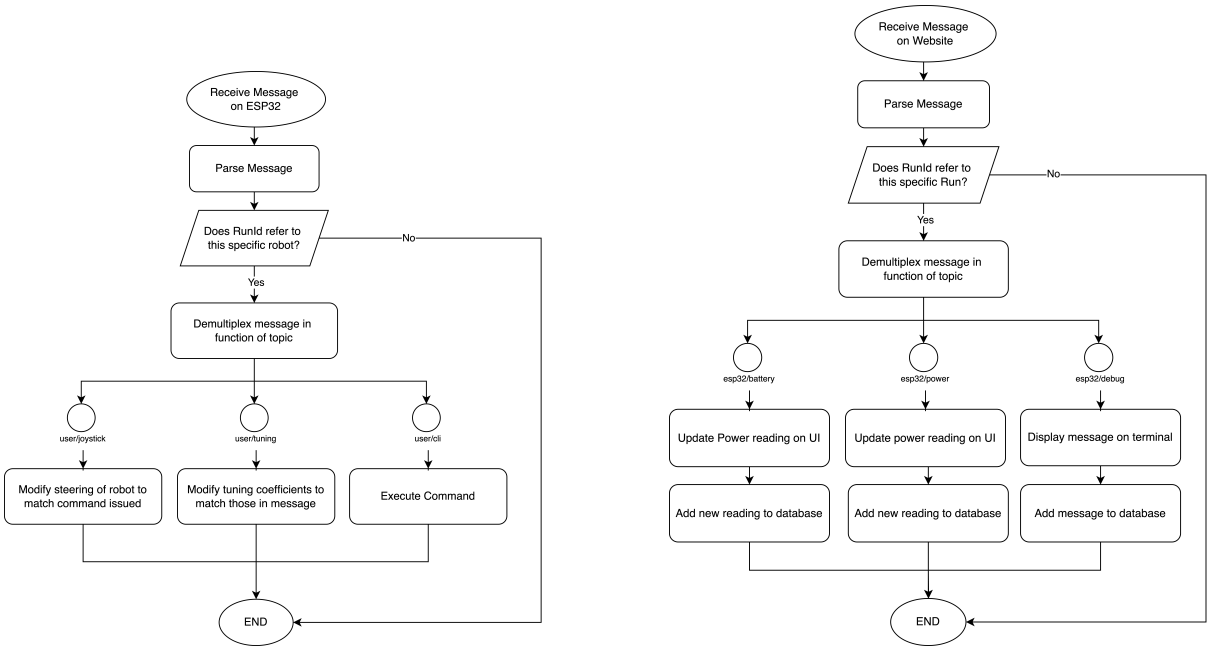
7.4.3 Receiving Messages

A message needs to be processed as soon as it is received. The processing of an MQTT message depends on its topic. A comprehensive summary of these different processings can be found in Figure 54.

7.5 Database Design

7.5.1 Implementing with DynamoDB

As MQTT messages are received by the broker, new information about the runs is made available to the user. That information should be saved to a database for future reference. SQL's rigid structure and familiarity was initially appealing. However, further research led to a migration to NoSQL for its low-latency and high-performance [35]. DynamoDB was eventually chosen for its seamless integration with AWS services [36].



(a) On ESP32 (b) On Website

Figure 54: Processing of Messages Received on Different Platforms

A Python script is run as a *Daemon* on the EC2 instance. This script dynamically saves MQTT messages to the database as per Figure 54. The architecture used for this end is underlined in Figure 55.

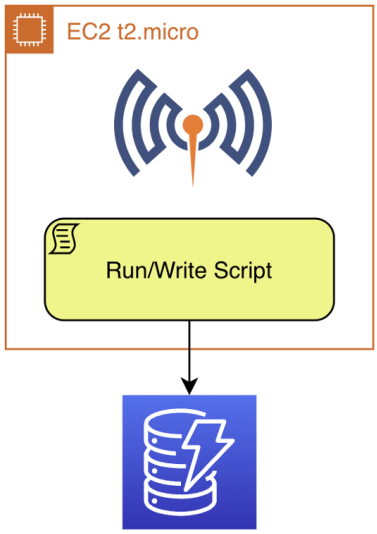


Figure 55: Architecture with DynamoDB

7.5.2 Designing for Queries

Data pertaining to each run had to be saved to the database as it was received for eventual fetching. This sections deals with the structure chosen to optimally save that data for future queries.

Schema flexibility is inherent to NoSQL databases. Databases can then be designed around and optimised for specific queries. This optimization involves choosing an optimal unique identifier for each item, which comes in the form of a partition key and a sort key. The partition key is crucial for evenly distributing data across partitions [37]. The sort key, on the other hand, enables efficient querying and range-based scans within each partition [38].

Partitions keys were then used to differentiate between different runs through *RunID*. The sort key *Type-Timestamp* was used to comb through the different data types (battery reading, power reading and debugging messages). Timestamp was added in that context to uniquely identify an entry from the other in the table.

This approach allowed for efficient fetching of the targeted data. For example, fetching all battery readings is made easy by setting *RunID* to a specific value, then looking up the sort key starting with *Battery*. Table 9 below outlines our database design for these fetching queries.

Partition Key	Sort Key	Attributes				
RunID	Type-{Timestamp}	Timestamp	Battery	Power	Message	MessageType
int	Battery-{Timestamp}	int	int	-	-	-
int	Power-{Timestamp}	int	-	int	-	-
int	Debug-{Timestamp}	int	-	-	string	received/sent

Table 9: NoSQL table for saving Run data

For additional examples, see Appendix E.

7.5.3 Testing and Improvements

Testing DynamoDB showed average write time of around 200ms, causing program stalls and missed MQTT messages. To address this, threads were added for write operations to run concurrently with the main program (listing 10). This approach allowed for improved efficiency and application responsiveness by ensuring no MQTT messages are missed during high-load periods.

```
from concurrent.futures import ThreadPoolExecutor

# ThreadPoolExecutor for handling database operations
executor = ThreadPoolExecutor(max_workers=10)

# Distribute message to correct function for further processing
def on_message(client, userdata, msg):
    if msg.topic == "esp32/battery":
        process_battery(msg)
    # more code ...

# Handle battery message
def process_battery(msg):
    try:
        data = json.loads(msg.payload.decode())
        run_id = data['run_id']
        timestamp = data['timestamp']
        battery = data['battery']

        # Submit the database operation to the ThreadPoolExecutor
        executor.submit(create_battery_entry, run_id, timestamp, battery)
```

Listing 10: Database writes handled on concurrent threads

7.6 Integration with Lambda Functions

Whenever dashboard is opened, previous battery readings need to be fetched from the database. This task is infrequent, but requires bursts of performance, a fitting use-case for leveraging Lambda functions' auto-scaling capabilities [39] [40].

These Lambda functions are event driven, and can triggered by HTML GET and POST functions through AWS GateWay. The architecture used for our ends is described in Figure 56.

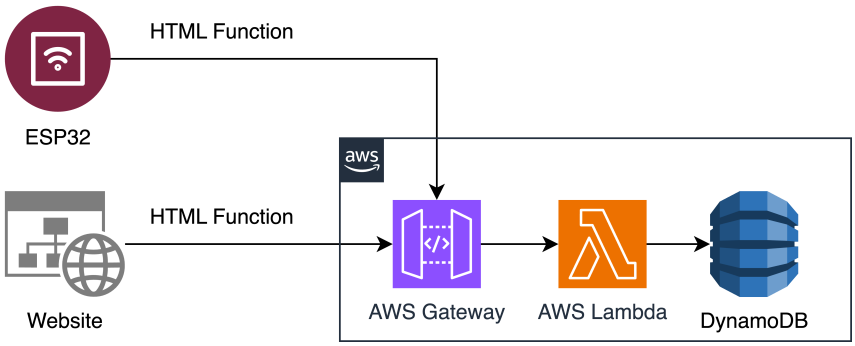


Figure 56: Architecture for integration of Lambda functions with DynamoDB

7.6.1 Using Lambda Functions to Build Dynamic Graphs

A Lambda function fetching previous battery reading could be used in conjunction with live MQTT message processing to dynamically build a graph, as can be seen in Figure 57.

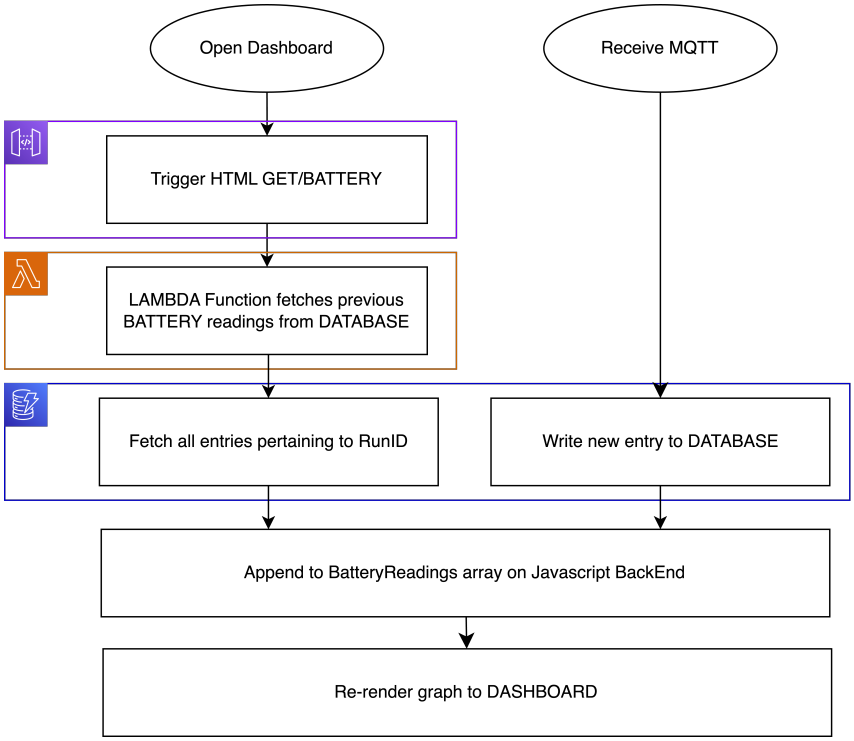


Figure 57: Flowchart for Battery Time Evolution Graphs

Another feature using Lambda functions can be found in Appendix E.

7.6.2 Testing

In order to test the Lambda functions we have written, HTTP GET and POST were sent to the appropriate API using POSTMAN, (Figure 58).

The average fetch time for 100 database entries was measured to be 1.257s.

7.7 Future Scopes

An initial outline for mapping has been designed to run on localhost. As of now, PGM and YAML files are transferred to a local host using Secure Shell SSH. A Python code then outputs a PNG file used in the UI.

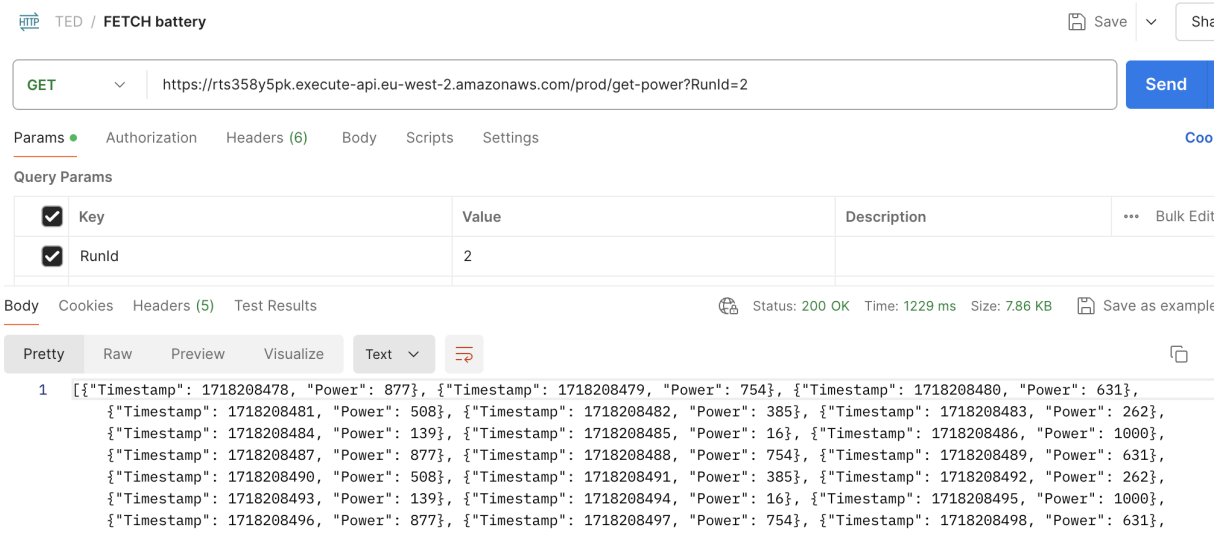
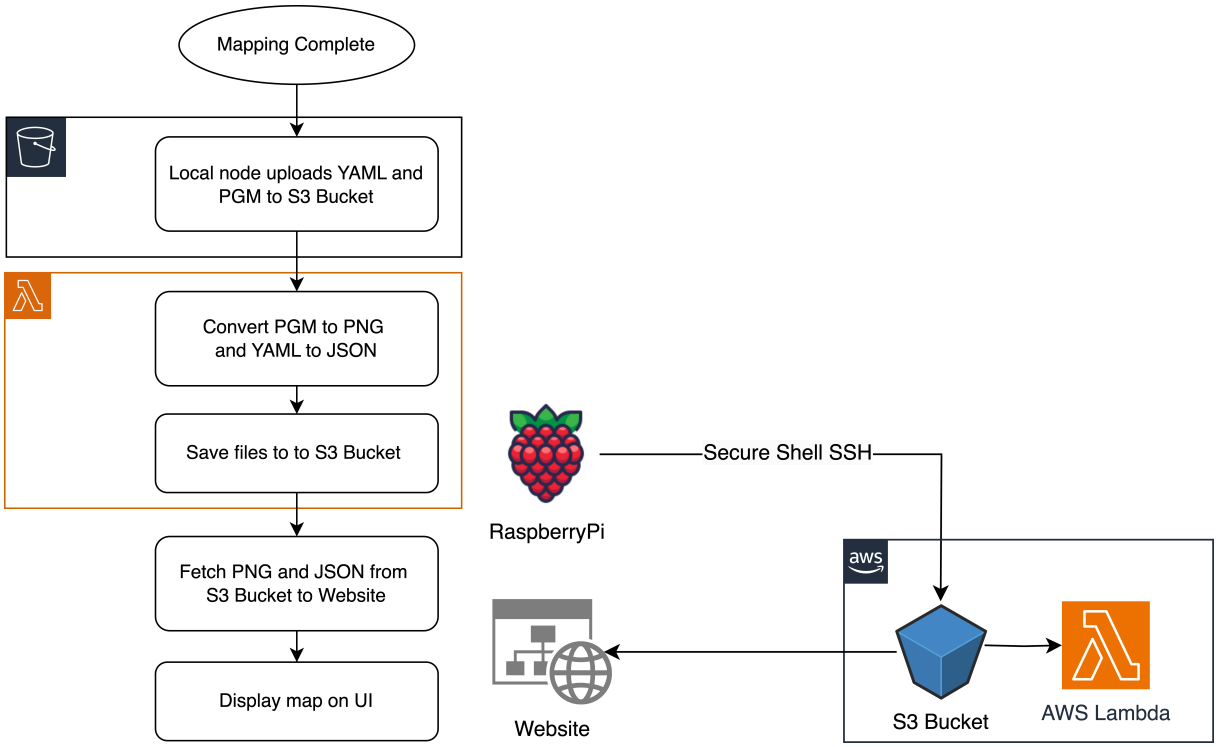


Figure 58: Testing Lambda Functions with POSTMAN

This task could also be automated using Lambda functions. S3 buckets, which act like file containers, could trigger a Lambda function whenever new files are received. Figure 59 outlines the flow data we have implemented in that spirit.



(a) Flowchart for mapping using AWS services

(b) S3 Bucket and Lambda functions for mapping

Figure 59: Plans for future implementation of mapping

7.8 System Testing

Integration testing was conducted, verifying the functionality and decoupling of various components of the ESP32 and website interactions. This process involved simultaneously running multiple ESP32 devices over varying run-times, each device continuously sending data to and receiving data from the website, using the testing methodology outlined earlier in Figure 3.

This rigorous testing was integrated throughout the architecture design phase. The overall user-experience was refined continuously, as per Figure 3.

8 Embedded

8.1 Requirements

The ESP32 microcontroller is responsible for concurrently executing the following tasks:

1. Responding to user input.
2. Stabilizing the bot.
3. Monitoring the power.

These tasks must operate simultaneously with minimal mutual interference. Consequently, the following non-functional requirements were drawn:

1. The tasks must use less than 160 KB of RAM, as the ESP32 has 160 KB of statically allocated DRAM (data RAM) available [41].
2. The bot must continue to be stabilised when the Wi-Fi signal drops.

8.2 System Architecture

To make the ESP32 code testable and maintainable, the system architecture was designed to decouple the system to the greatest extent possible. This approach is described in detail in Figure 60.

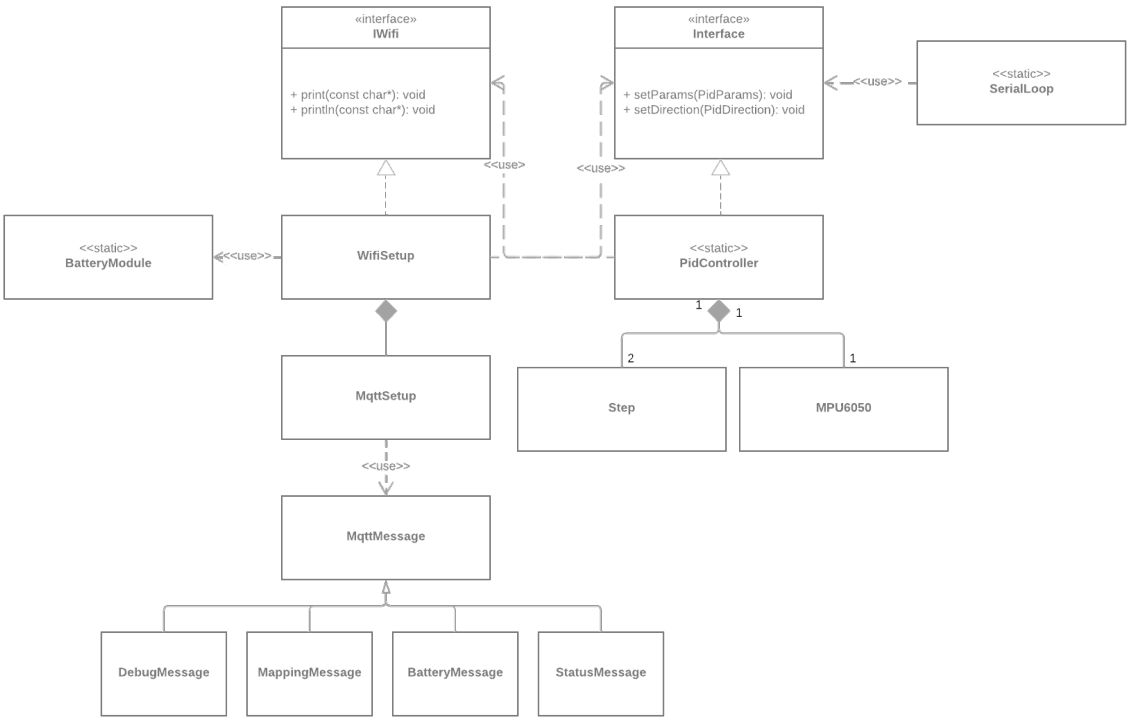


Figure 60: ESP32 UML Class Diagram

8.3 Concurrency and Parallelism

The ESP32 is equipped with FreeRTOS, allowing every task to be programmed as if it were its own task by using the function `xTaskCreatePinnedToCore()`. The parameters for that function are shown in Table 10.

Some key observations:

- When given a priority of 1, `pidLoop` ran very slowly, which made the bot unstable. When the priority was increased, it returned to normal.

- The ESP32 is a dual core processor. Wi-Fi and system tasks use core 0 by default [42], therefore, core 0 was reserved for `wifiLoop`.
- Core 1 is reserved for `pidLoop`, to guarantee enough system resources for it.
- The total reserved stack space is 65 KB, using 41% of the total RAM.

Task Name	Stack Size (KB)	Priority	Core
wifiLoop	10	1	0
pidLoop	30	2	1
serialLoop	10	1	0/1
batteryLoop	15	1	0/1

Table 10: Task Parameters

8.3.1 Preventing Race Conditions

When an instruction to write the PID parameters comes from `wifiLoop`, `pidLoop` could be trying to read it at the same time.

This is known as a data race, an undefined behaviour: the read may return a "half-written" value, combining the values from before and after the write [43]. To prevent this, we can use a mutex, ensuring that only one thread accesses a specific resource at a time. This concept is illustrated in Figure 61. The code to write the parameters is shown for reference in Listing 11, with the code to read being almost identical.

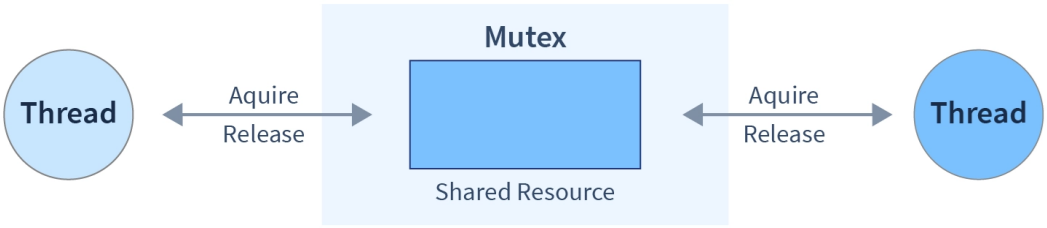


Figure 61: Mutex Internals [44]

```
1  void PidController::setParams(PidParams params)
2  {
3      // Take mutex and wait forever until it is available
4      xSemaphoreTake(paramsMutex, portMAX_DELAY);
5      PidController::params = params;
6      xSemaphoreGive(paramsMutex);
7  }
```

Listing 11: Function to set the PID parameters (from `PidController.cpp`)

8.3.2 Preventing Loss of Control

Upon loss of Wi-Fi signal, the bot should automatically assume zero movement to avoid potentially crashing into a wall. There is a fail-safe built into `wifiLoop`, which stops the robot when the server goes down (see `MqttSetup.cpp`).

8.4 Testing

8.4.1 Metholodogy

Automated tests were written for the ESP32 using the PlatformIO test framework (Unity). Tests could either be ran on the ESP32 or natively.

All the tasks in Table 10 have tests written for them as seen on the GitHub repository (see Appendix A). Figure 62 shows the test results, with 2 tests running on the ESP32 and 2 tests running natively. Running native tests were preferred, as they would run over 10 times faster.

```
saturn691@saturn:~/BalanceBot/esp32$ pio test
Verbosity level can be increased via `-v, -vv, or -vvv` option
Collected 4 tests

Processing test_mpu6050 in esp32doit-devkit-v1 environment
-------------------------------------------------------------------------------------------------------------------
Building & Uploading...
Testing...
If you don't see any output for the first 10 secs, please reset board (press reset button)

test/test_mpu6050/test_mpu6050.cpp:39: test_accelerometer_values        [PASSED]
----------------------- esp32doit-devkit-v1:test_mpu6050 [PASSED] Took 17.99 seconds -----------------------

Processing test_wifi in esp32doit-devkit-v1 environment
-------------------------------------------------------------------------------------------------------------------
Building & Uploading...
Testing...
If you don't see any output for the first 10 secs, please reset board (press reset button)

test/test_wifi/test_wifi.cpp:125: test_wifi_connect     [PASSED]
test/test_wifi/test_wifi.cpp:126: test_mqtt_connect     [PASSED]
test/test_wifi/test_wifi.cpp:127: test_wifi_disconnect  [PASSED]
test/test_wifi/test_wifi.cpp:128: test_remote_joystick  [PASSED]
test/test_wifi/test_wifi.cpp:129: test_remote_pid       [PASSED]
----------------------- esp32doit-devkit-v1:test_wifi [PASSED] Took 42.05 seconds -----------------------

Processing test_battery in native environment
-------------------------------------------------------------------------------------------------------------------
Building...
Testing...
test/test_battery/test_battery.cpp:30: test_battery     [PASSED]
------------------------------ native:test_battery [PASSED] Took 0.49 seconds ------------------------------

Processing test_serial in native environment
-------------------------------------------------------------------------------------------------------------------
Building...
Testing...
test/test_serial/test_serial.cpp:51: test_decode        [PASSED]
------------------------------ native:test_serial [PASSED] Took 0.41 seconds ------------------------------

================================================== SUMMARY ==================================================
Environment          Test          Status    Duration
-------------------  ------------  --------  ------------
esp32doit-devkit-v1  test_mpu6050  PASSED    00:00:17.990
esp32doit-devkit-v1  test_wifi     PASSED    00:00:42.052
native               test_battery  PASSED    00:00:00.491
native               test_serial   PASSED    00:00:00.406
================================ 8 test cases: 8 succeeded in 00:01:00.939 =================================
```

Figure 62: Full console output of automated testing

8.4.2 Continuous Integration

Furthermore, when code is committed to the repository, the tests shown in Figure 62 are ran automatically with GitHub Actions. An old computer connected to an ESP32, as seen in Figure 63, also had to be setup as a "runner" [45] to allow embedded tests to be ran. Figure 64 shows the result of the CI test.

8.5 Evaluation

By using FreeRTOS and allocating a reasonable amount of system resources, the functional requirements (tasks 1-3) are satisfied.

Since we are using less than 160 KB of RAM (65 KB), the first non-functional requirement is satisfied.

By putting a fail-safe and by reserving a core for the `pidLoop`, the second non-functional requirement is satisfied.

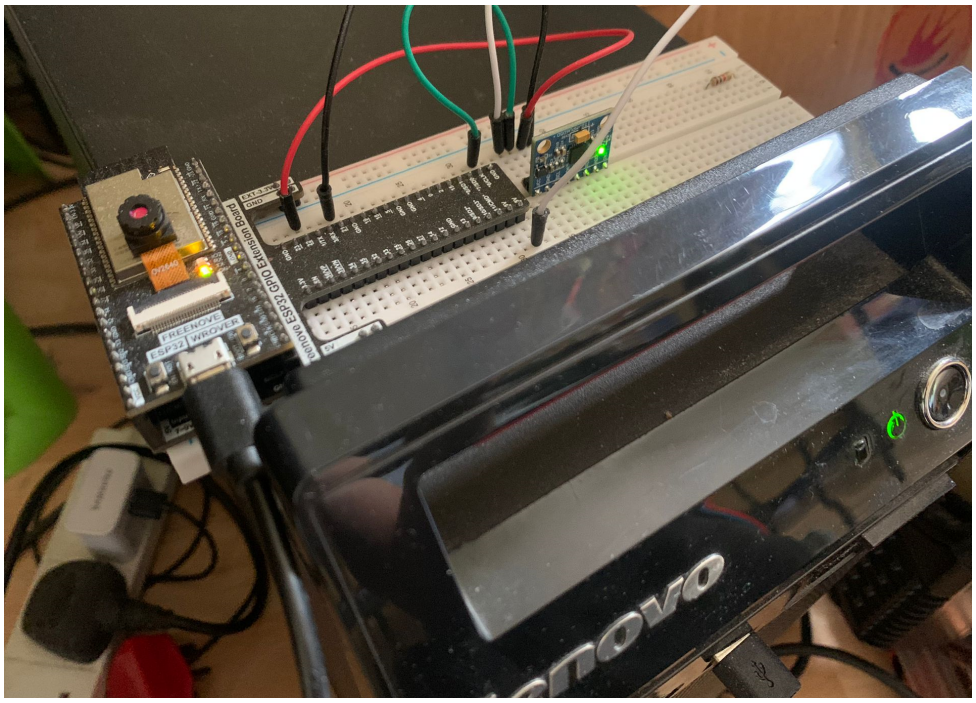


Figure 63: CI server setup

Figure 64: Automatic testing of the ESP32 (green tick)

9 Conclusion

9.1 Top-Level Testing and Evaluation

Our self-balancing bot successfully traversed the arena shown in Figure 65 and outputted the UI shown in Figure 66. It was able to respond to the incident, characterized by the green box and report it. The integrated system has met the requirements outlined in the introduction.

Figure 65: Test Arena

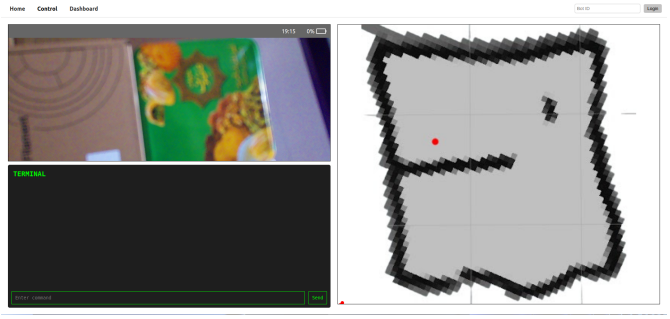


Figure 66: User Interface

9.2 Cost Breakdown

We managed to stay within budget for the project, taking advantage of 'open-source' components available through the college's robotics lab. Total costs of the project are broken down in Appendix B.

9.3 Future Work

Considering the short time frame of the project, there are many avenues to pursue in terms of future work. Some potential improvements include:

1. **Automatic Rebalancing**

   A mechanical arm could be used to correct and re-balance the robot in case it falls over.

2. **Head unit protection**

   A helmet or more resilient head unit would protect the wire connections on the ESP32 and Raspberry Pi in case of malfunction.

3. **Live Camera Feed**

   A live camera feed provides more information to the user; the user would have the option to observe how the rover is traversing through new terrain.

4. **Processing of LiDAR on Server**

   This lays the groundwork for a centralised unit to handle all system processing.

5. **Multiple Incident Detection**

   Currently, the system only detects one type of incident (over-turned crates) out of the many discussed in the introduction. This can be extended through research and implementation of more advanced computer vision techniques.

6. **Battery Management Circuits with Better Noise Resistance**

   The battery management circuit can be improved by using resistors with lower tolerances and introducing further measures to reduce the effect of noise on the system.

7. **Development of a Precise Robot Model**

   An accurate model will allow for further development of optimal control algorithms such as LQR, MPC, and precise tuning through neural networks.

9.4 Reflection

Throughout this project, we gained valuable insights and experience in several areas:

- **Communication and Teamwork:** Regular meetings were held to ensure all team-members were up to data on progress made in different subsections. These meetings set the benchmark for all the subsystems, and helped define the interfaces between these subsystems.
- **Adaptability and Resourcefulness:** Unforeseen circumstances were limited by planning around them (including FMEA), recalculating priorities as needed.
- **Project Management and Planning:** Organising the tasks through the Agile methodology helped us understand the importance of retrospection and milestones planning.

References

[1] Atlassian. "What is the agile methodology?" (), [Online]. Available: https://www.atlassian.com/agile.

[2] Wrike. "What is a sprint in agile?" (), [Online]. Available: https://www.wrike.com/project-management-guide/faq/what-is-a-sprint-in-agile/.

[3] Gadgetoid. "Uart at raspberry pi gpio pinout." (), [Online]. Available: https://pinout.xyz/pinout/uart.

[4] U. of Michigan. "Inverted pendulum: System modeling." (), [Online]. Available: https://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling.

[5] Krishnavedala. "Schematic drawing of an inverted pendulum on a cart." (), [Online]. Available: https://en.wikipedia.org/wiki/File:Cart-pendulum.svg.

[6] JavatPoint. "Pid controller c++." (), [Online]. Available: https://www.javatpoint.com/pid-controller-cpp.

[7] J. Bennett, A. Bhasin, J. Grant, and W. C. Lim, "Pid tuning via classical methods," M.S. thesis, University of Michigan.

[8] M. Alam. "Measure tilt angle using mpu6050 gyro/accelerometer & arduino." (), [Online]. Available: https://how2electronics.com/measure-tilt-angle-mpu6050-arduino/.

[9] Linkoping, *Angle estimation using gyros and accelerometers*. [Online]. Available: https://www.control.lth.se/fileadmin/control/Education/EngineeringProgram/FRTF05_China/pm_sensor_revised.pdf.

[10] LastMinuteEngineering. "Control stepper motor with a4988 driver module & arduino." (), [Online]. Available: https://lastminuteengineers.com/a4988-stepper-motor-driver-arduino-tutorial/.

[11] P. P. "Pid controllers." (), [Online]. Available: https://tttapa.github.io/Pages/Arduino/Control-Theory/Motor-Fader/PID-Controllers.html.

[12] S. Bajonczak. "Mastering the measurement of battery voltage with an esp controller." (), [Online]. Available: https://blog.bajonczak.com/measure-voltage-with-an-esp-controller/.

[13] C. Atwell. "Boost those weak voltages with op amps." (), [Online]. Available: https://www.electronicdesign.com/technologies/analog/article/21245552/boost-those-weak-voltages-with-op-amps.

[14] K. Srinivasan. "Better pmic design using multi-physics simulation." (), [Online]. Available: https://semiengineering.com/better-pmic-design-using-multi-physics-simulation/.

[15] DevXplained. "Diodes as input protection." (), [Online]. Available: https://devxplained.eu/en/blog/diodes-as-input-protection.

[16] Ashish. "What are the different methods to estimate the state of charge of batteries?" (), [Online]. Available: https://www.scienceabc.com/innovation/what-are-the-different-methods-to-estimate-the-state-of-charge-of-batteries.html.

[17] Energizer, *Nickel metal hydride application manual*. [Online]. Available: https://data.energizer.com/pdfs/nickelmetalhydride_appman.pdf.

[18] A. Pasescu. "Ldrobot ld06 360 lidar module & raspberry pi mounting bracket." (), [Online]. Available: https://grabcad.com/library/ldrobot-ld06-360-lidar-module-raspberry-pi-mounting-bracket-1.

[19] The Cartographer Authors Revision. "Algorithm walkthrough for tuning." (), [Online]. Available: https://google-cartographer-ros.readthedocs.io/en/latest/algo_walkthrough.html.

[20] Awada. "Autonomous mobile robot (amr): Focus on autonomous exploration." (), [Online]. Available: https://awabot.com/wp-content/uploads/2021/09/frontieres_anglais-300x280.png.

[21] MathWorks. "Pure pursuit controller." (), [Online]. Available: https://www.mathworks.com/help/nav/ug/pure-pursuit-controller.html.

[22] M. Dakulovic. "Two-way d* algorithm for path planning and replanning." (), [Online]. Available: https://www.semanticscholar.org/paper/Two-way-D*-algorithm-for-path-planning-and-Dakulovic-Petrovi%C4%87/3c7f35cba819898c7939f13da2cf51b97092abcb/figure/2.

[23] BBC, "Meet the great british bake off illustrator: Tom hovey," [Online]. Available: https://www.bbc.co.uk/programmes/articles/4fr6jXM3XQpXG9sc8t0tQH/meet-the-great-british-bake-off-illustrator-tom-hovey.

[24] AWS, "What is mqtt?," [Online]. Available: https://aws.amazon.com/what-is/mqtt/.

[25] HiveMQ. "Mqtt essentials." (), [Online]. Available: https://www.hivemq.com/mqtt/.

[26] E. Bruno. "Introducing the constrained application protocol (coap)." (), [Online]. Available: https://blogs.oracle.com/javamagazine/post/java-coap-constrained-application-protocol-introduction.

[27] Digi. "What is zigbee?" (), [Online]. Available: https://www.digi.com/solutions/by-technology/zigbee-wireless-standard.

[28] MQTT. "Overview of mqtt." (), [Online]. Available: https://mqtt.org/.

[29] "What is mqtt and how it works?" (), [Online]. Available: https://randomnerdtutorials.com/what-is-mqtt-and-how-it-works/.

[30] AWS, "Aws iot core documentation," [Online]. Available: https://docs.aws.amazon.com/iot/.

[31] Eclipse. "An open source mqtt broker." (), [Online]. Available: https://mosquitto.org/.

[32] AWS. [Online]. Available: https://calculator.aws/#/createCalculator/IoTCore.

[33] C. Burgdorfer. "How to set up a mosquitto mqtt broker securely." (), [Online]. Available: https://medium.com/gravio-edge-iot-platform/how-to-set-up-a-mosquitto-mqtt-broker-securely-using-client-certificates-82b2aaaef9c8.

[34] MqttJmeter. "Mqtt-jmeter overview." (), [Online]. Available: https://github.com/emqx/mqtt-jmeter.

[35] D. Flynn. "Iot considerations — storage and database, sql, nosql, historical data." (), [Online]. Available: https://medium.com/lattice-research/iot-considerations-storage-and-database-sql-nosql-historical-data-1b9e4d53cb32..

[36] AWS, "Nosql design for dynamodb," [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html.

[37] AWS, "Choosing the right dynamodb partition key," [Online]. Available: https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/.

[38] AWS, "Best practices for using sort keys to organize data," [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-sort-keys.html.

[39] AWS, "What is aws lambda?," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[40] AWS, "Using aws lambda with amazon dynamodb," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/with-ddb.html.

[41] Espressif Systems (Shanghai) Co., Ltd. "Memory types." (), [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/memory-types.html.

[42] Espressif Systems (Shanghai) Co., Ltd. "Speed optimization." (), [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/performance/speed.html.

[43] Wikipedia. "Race condition." (), [Online]. Available: https://en.m.wikipedia.org/w/index.php?title=Race_condition%5C&diffonly=true#Data_race.

[44] A. Raj. "Mutex in os." (), [Online]. Available: https://www.scaler.com/topics/operating-system/mutex-in-os/.

[45] GitHub, Inc. "About self-hosted runners." (), [Online]. Available: https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners.

[46] Burr-Brown Corporation. "Single-supply, micropower operational amplifiers." (), [Online]. Available: https://www.ti.com/lit/ds/symlink/opa2241.pdf?ts=1718639316854&ref_url=https%253A%252F%252Fwww.mouser.com%252F.

Appendices

A GitHub Repo - BalanceBot

https://github.com/TorturedEngineersDept/BalanceBot

B Budget

We were given a budget of £60.00 to abide by. The team put great emphasis on abiding by that budget. The table below highlights these efforts.

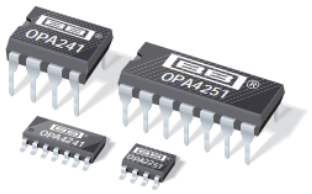
Item	Part No	Unit Price	Qty	Price	Source
LiDAR	LD06	£40.86	1	£40.86	RS Electronics
Op Amps	OPA2241PA	£4.55	4	£18.20	RS Electronics
Raspberry Pi 4		*borrowed*	1	£0	IC Robotics Society
Camera	RPi Camera Module 2	*borrowed*	1	£0	IC Robotics Society
			TOTAL	£59.06	

C FMEA Analysis of Autonomous Component

Process Purpose	Potential Failure Mode	Severity	Potential Causes of Failure	Occurrence	Process Control	Detection	RPN	Recommended Action	Severity	Occurrence	Detection	New RPN
Sensing	Sensor disconnection	8	Loose connections, component failure	4	Alert user in case disconnection	1	32	None				
Motor	Wheel slipping	2	Loose connections, unsupported terrain	4	None	10	80	None				
	Motor malfunction	9	Wear and tear, overheating	5	Current limiting, test for motor disconnection	1	45	None				
WiFi	Loss of connection/loss of position	10	Weak signal, sudden network failure	3	Automatically set control input to 0 upon loss of connection	1	30	Alert user of poor signal on reconnection	7	3	1	21
Power	Battery failure	10	Battery degradation, over-discharge	3	2A fuse in place	1	30	None				
Balancing/ Tilt Control	Extreme oscillations	6	Sensor drift, extreme disturbances	6	Fine-tuning of PID, implementation of tilt angle limits	3	108	Implementation of model predicitive control (MPC)	6	2	3	36
Lateral Movement/Velocity Control	Inconsistent velocity	2	Unforseen surfaces, sensor drift	8	None	10	160	Clearly communicate supported surfaces/inclines to the user	2	3	10	60
Chassis	Loose screws/nuts	5	Wear and tear	2	Multiple mounts for each joint	10	100	Use 'loctite'	5	1	10	50
	Chassis Damage	5	Collisions, material wear and tear	4	Manual override of control to prevent collisions	5	100	None				
Rotation/Yaw Control	Sensor drift	4	Calibration issues, environmental disturbances e.g. wind	5	None	10	200	Use sensor with magnetometer for self-calibration	4	1	3	12
Embedded	Incorrect PID gains	9	Read and write PID coefficients at the same time	2	Implementation of mutex (section 8)	1	18	None				

D OPA2241PA [46]

BURR-BROWN®



OPA241
OPA2241
OPA4241

OPA251
OPA2251
OPA4251

Single-Supply, *Micro*POWER OPERATIONAL AMPLIFIERS

OPA241 Family optimized for +5V supply.

OPA251 Family optimized for ±15V supply.

FEATURES

- ***Micro*POWER: I_Q = 25µA**
- **SINGLE-SUPPLY OPERATION**
- **RAIL-TO-RAIL OUTPUT (within 50mV)**
- **WIDE SUPPLY RANGE**
  Single Supply: +2.7V to +36V
  Dual Supply: ±1.35V to ±18V
- **LOW OFFSET VOLTAGE: ±250µV max**
- **HIGH COMMON-MODE REJECTION: 124dB**
- **HIGH OPEN-LOOP GAIN: 128dB**
- **SINGLE, DUAL, AND QUAD**

APPLICATIONS

- **BATTERY OPERATED INSTRUMENTS**
- **PORTABLE DEVICES**
- **MEDICAL INSTRUMENTS**
- **TEST EQUIPMENT**

DESCRIPTION

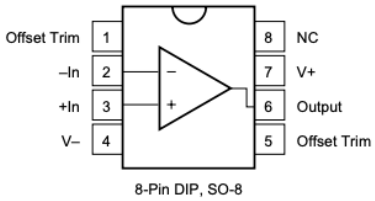
The OPA241 series and OPA251 series are specifically designed for battery powered, portable applications. In addition to very low power consumption (25µA), these amplifiers feature low offset voltage, rail-to-rail output swing, high common-mode rejection, and high open-loop gain.

The OPA241 series is optimized for operation at low power supply voltage while the OPA251 series is optimized for high power supplies. Both can operate from either single (+2.7V to +36V) or dual supplies (±1.35V to ±18V). The input common-mode voltage range extends 200mV below the negative supply—ideal for single-supply applications.

They are unity-gain stable and can drive large capacitive loads. Special design considerations assure that these products are easy to use. High performance is maintained as the amplifiers swing to their specified limits. Because the initial offset voltage (±250µV max) is so low, user adjustment is usually not required. However, external trim pins are provided for special applications (single versions only).

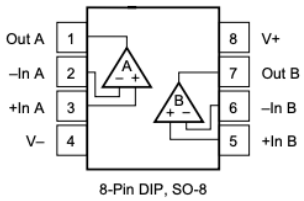
The OPA241 and OPA251 (single versions) are available in standard 8-pin DIP and SO-8 surface-mount packages. The OPA2241 and OPA2251 (dual versions) come in 8-pin DIP and SO-8 surface-mount packages. The OPA4241 and OPA4251 (quad versions) are available in 14-pin DIP and SO-14 surface-mount packages. All are fully specified from –40°C to +85°C and operate from –55°C to +125°C.

OPA241, OPA251



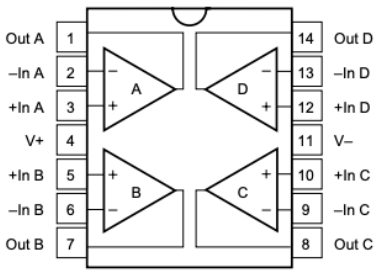
8-Pin DIP, SO-8

OPA2241, OPA2251



8-Pin DIP, SO-8

OPA4241, OPA4251



14-Pin DIP, SO-14

International Airport Industrial Park • Mailing Address: PO Box 11400, Tucson, AZ 85734 • Street Address: 6730 S. Tucson Blvd., Tucson, AZ 85706 • Tel: (520) 746-1111 • Twx: 910-952-1111
Internet: http://www.burr-brown.com/ • FAXLine: (800) 548-6133 (US/Canada Only) • Cable: BBRCORP • Telex: 066-6491 • FAX: (520) 889-1510 • Immediate Product Info: (800) 548-6132

SPECIFICATIONS: V_S = ±15V

At T_A = +25°C, R_L = 100kΩ connected to ground, unless otherwise noted.
Boldface limits apply over the specified temperature range, T_A = –40°C to +85°C.

PARAMETER		CONDITION	OPA241UA, PA OPA2241UA, PA OPA4241UA, PA			OPA251UA, PA OPA2251UA, PA OPA4251UA, PA			UNITS
			MIN	TYP	MAX	MIN	TYP	MAX	
OFFSET VOLTAGE									
Input Offset Voltage	V_{OS}			±100			±50	±250	µV
T_A = –40°C to +85°C				**±150**			**±100**	**±300**	µV
vs Temperature	dV_{OS}/dT	**T_A = –40°C to +85°C**		**±0.6**			**±0.5**		µV/°C
vs Power Supply	PSRR	V_S = ±1.35V to ±18V		*	*		3	30	µV/V
T_A = –40°C to +85°C		V_S = ±1.35V to ±18V			*			**30**	µV/V
Channel Separation (dual, quad)				*			0.3		µV/V
INPUT BIAS CURRENT									
Input Bias Current[1]	I_B			*			–4	–20	nA
T_A = –40°C to +85°C								**–25**	nA
Input Offset Current	I_{OS}			*			±0.1	±2	nA
T_A = –40°C to +85°C								**±2**	nA
NOISE									
Input Voltage Noise, f = 0.1Hz to 10Hz				*			1		µVp-p
Input Voltage Noise Density, f = 1kHz	e_n			*			45		$nV/\sqrt{Hz}$
Current Noise Density, f = 1kHz	i_n			*			40		$fA/\sqrt{Hz}$
INPUT VOLTAGE RANGE									
Common-Mode Voltage Range	V_{CM}					(V–) –0.2		(V+) –0.8	V
Common-Mode Rejection Ratio	CMRR	V_{CM} = –15.2V to 14.2V		*		100	124		dB
T_A = –40°C to +85°C		V_{CM} = –15V to 14.2V				**100**			dB
INPUT IMPEDANCE									
Differential				*			10^7 || 2		Ω || pF
Common-Mode				*			10^9 || 4		Ω || pF
OPEN-LOOP GAIN									
Open-Loop Voltage Gain	A_{OL}	R_L = 100kΩ, V_O = –14.75V to +14.75V		*		100	128		dB
T_A = –40°C to +85°C		R_L = 100kΩ, V_O = –14.75V to +14.75V				**100**			dB
		R_L = 20kΩ, V_O = –14.7V to +14.7V		*		100	128		dB
T_A = –40°C to +85°C		R_L = 20kΩ, V_O = –14.7V to +14.7V				**100**			dB
FREQUENCY RESPONSE									
Gain-Bandwidth Product	GBW			*			35		kHz
Slew Rate	SR	G = 1		*			0.01		V/µs
Overload Recovery Time		V_{IN} • G = V_S		*			60		µs
OUTPUT									
Voltage Output Swing from Rail[2]	V_O	R_L = 100kΩ, A_{OL} ≥ 70dB		*			50		mV
		R_L = 100kΩ, A_{OL} ≥ 100dB		*			75	250	mV
T_A = –40°C to +85°C		R_L = 100kΩ, A_{OL} ≥ 100dB						**250**	mV
		R_L = 20kΩ, A_{OL} ≥ 100dB		*			100	300	mV
T_A = –40°C to +85°C		R_L = 20kΩ, A_{OL} ≥ 100dB						**300**	mV
Short-Circuit Current	I_{SC}								
Single Versions				*			–21/+4		mA
Dual Versions				*			–50/+4		mA
Capacitive Load Drive	C_{LOAD}			*			See Typical Curve		
POWER SUPPLY									
Specified Voltage Range	V_S			*			±15		V
Operating Voltage Range		**T_A = –40°C to +85°C**	*		*	**±1.35**		**±18**	V
Quiescent Current (per amplifier)	I_Q	I_O = 0		*			±27	±38	µA
T_A = –40°C to +85°C		I_O = 0						**±45**	µA
TEMPERATURE RANGE									
Specified Range			*		*	–40		+85	°C
Operating Range			*		*	–55		+125	°C
Storage Range			*		*	–55		+125	°C
Thermal Resistance	θ_{JA}								
8-Pin DIP				*			100		°C/W
SO-8 Surface Mount				*			150		°C/W
14-Pin DIP				*			80		°C/W
SO-14 Surface Mount				*			100		°C/W

* Specifications the same as OPA251UA, PA.

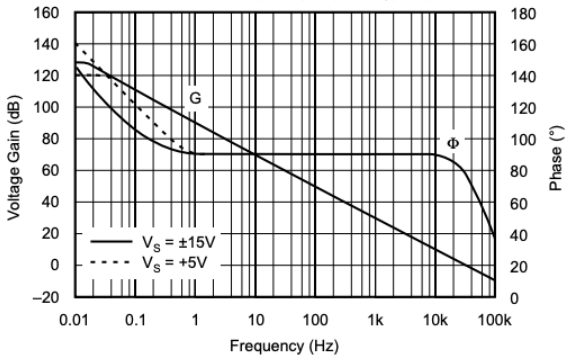
NOTES: (1) The negative sign indicates input bias current flows out of the input terminals. (2) Output voltage swings are measured between the output and power supply rails.

TYPICAL PERFORMANCE CURVES

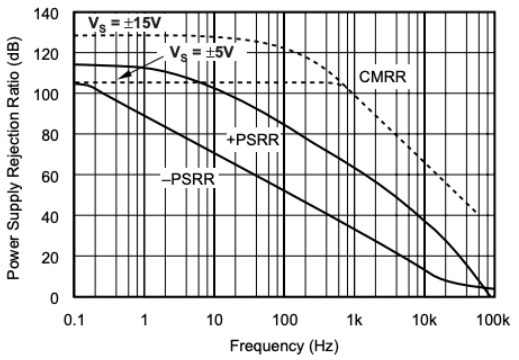
At T_A = +25°C, and R_L = 100kΩ connected to V_S/2 (ground for V_S = ±15V), unless otherwise noted.

Curves apply to OPA241 and OPA251 unless specified.

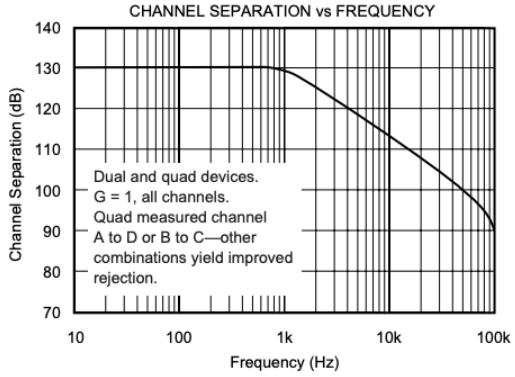
OPEN-LOOP GAIN/PHASE vs FREQUENCY



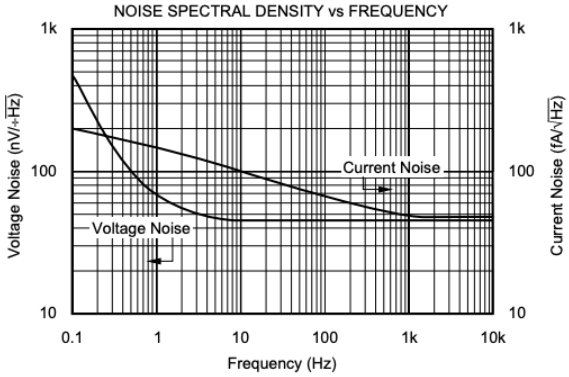
POWER SUPPLY and COMMON-MODE REJECTION RATIO vs FREQUENCY



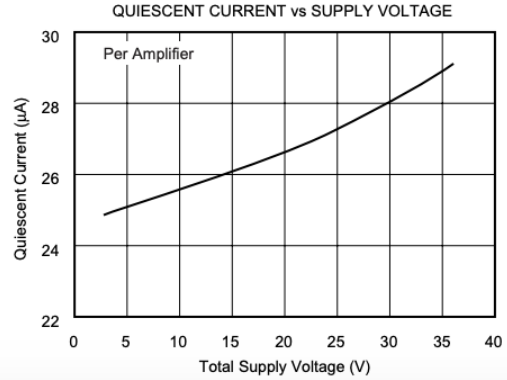
CHANNEL SEPARATION vs FREQUENCY



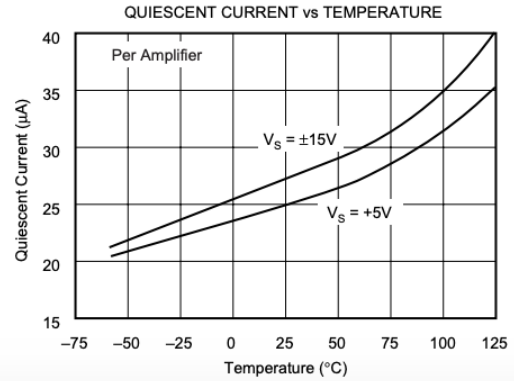
INPUT VOLTAGE AND CURRENT NOISE SPECTRAL DENSITY vs FREQUENCY



QUIESCENT CURRENT vs SUPPLY VOLTAGE



QUIESCENT CURRENT vs TEMPERATURE

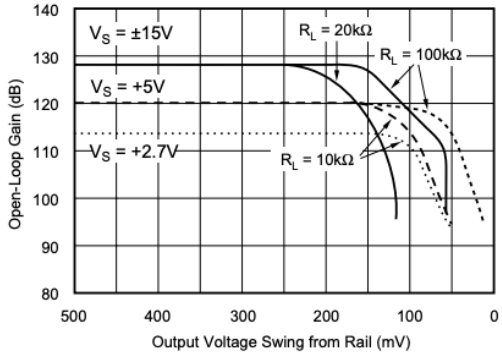


TYPICAL PERFORMANCE CURVES (CONT)

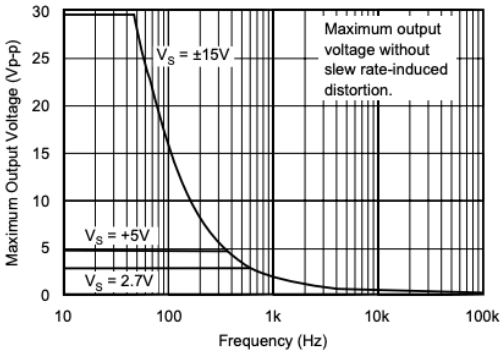
At T_A = +25°C, and R_L = 100kΩ connected to V_S/2 (ground for V_S = ±15V), unless otherwise noted.

Curves apply to OPA241 and OPA251 unless specified.

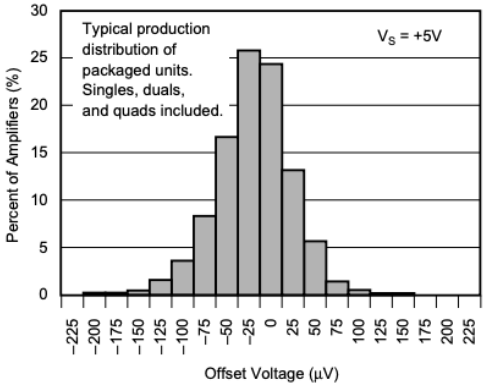
OPEN-LOOP GAIN vs OUTPUT VOLTAGE SWING



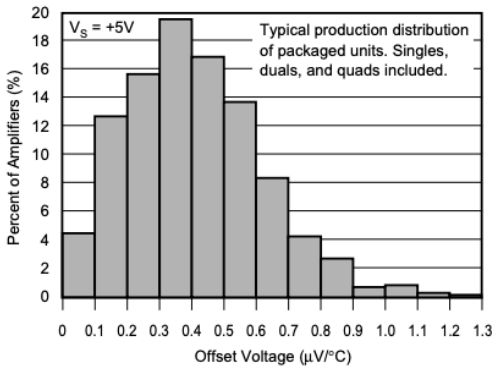
MAXIMUM OUTPUT VOLTAGE vs FREQUENCY



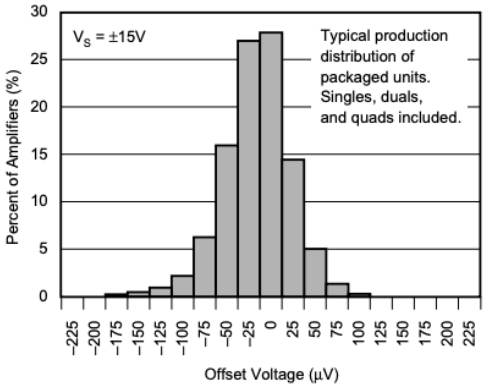
OPA241 SERIES OFFSET VOLTAGE PRODUCTION DISTRIBUTION



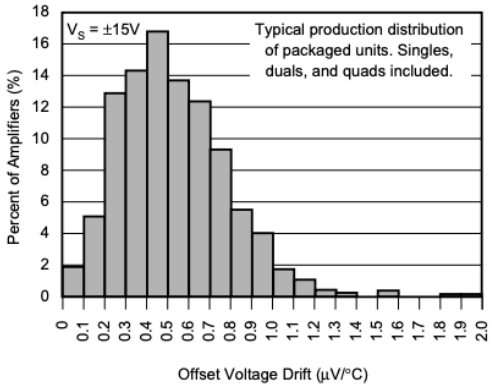
OPA241 SERIES OFFSET VOLTAGE DRIFT PRODUCTION DISTRIBUTION



OPA251 SERIES OFFSET VOLTAGE PRODUCTION DISTRIBUTION



OPA251 SERIES OFFSET VOLTAGE DRIFT PRODUCTION DISTRIBUTION



E Assigning Unique RunID

This sections aims to establish a framework allowing for robot/website pairs to communicate with each other with minimal interference.

This was made possible by assigning unique RunIDs.

Database Design for Unique RunID

Two criteria had to be simultaneously satisfied:

1. **Unique Identification of Robot** through a custom identifier BotId
2. **Association of BotId to RunId** to communicate to website

The NoSQL database then had to:

- Associate BotID to ESP32 Mac Addresses.
- Associate latest RunID to each BotID.
- Maintain a counter for new BotID and RunID.

The disposition in figure 67 was then chosen for our purposes.

BotID Table	
Partition Key	**Sort Key**
MACAddress (S)	BotId (N)

Counter Table	
Partition Key	**Attributes**
Counter (S)	Value (N)

RunID Table	
Partition Key	**Attributes**
BotId (N)	RunId (N)

Figure 67: DynamoDB Table Schema for RunId

Implementation on ESP32

Unique RunId were attributed whenever an ESP32 would connect to Wi-Fi. Indeed, a HTML GET would trigger a AWS Lambda function to fetch the desired RunId and BotId (figure 68a).

Implementation on Website

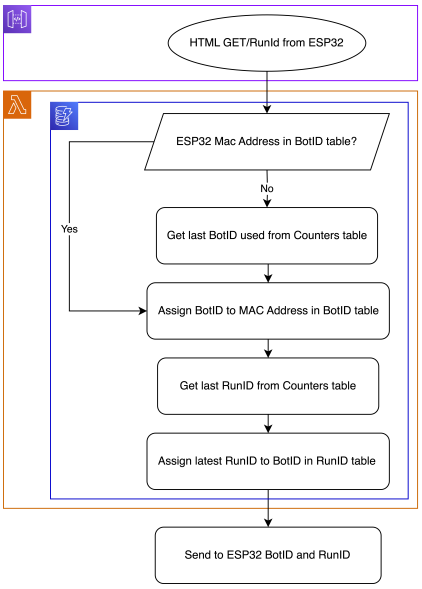
RunId would be obtained whenever the user would log into a new BotId. An HTML POST would then trigger an AWS Lambda function to query the database (figure 68b).

```
void WifiSetup::resolveId()
{
    HTTPClient http;
    String baseApiEndpoint =
    "https://rts358y5pk.execute-api.eu-west-2.amazonaws.com/prod/
    get-runid-esp32?mac=";

    // Construct the full API endpoint URL with the MAC address
    String macAddress = WiFi.macAddress();
    String apiEndpoint = String(baseApiEndpoint) + macAddress;

    http.begin(apiEndpoint);

    // Trigger Lambda function
    int httpResponseCode = http.GET();

    while (httpResponseCode != 200)
    {
        httpResponseCode = http.GET();
    }

    String response = http.getString();
    http.end();
```

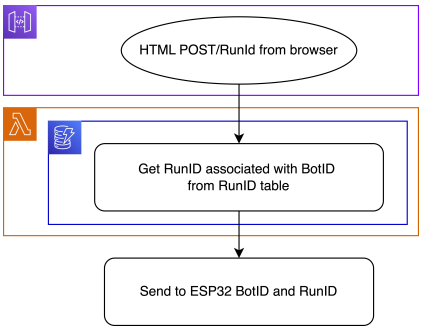
Listing 12: Function to get RunId for ESP32 (from `WifiSetup.cpp`)

```
export const handleLoginSubmit = async (botId) => {
    const data = { BotId: botId };

    try {
        const response = await fetch(
        'https://rts358y5pk.execute-api.eu-west-2.amazonaws.com/prod/
        get-runid-ui', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(data),
        });

        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const result = await response.json();
        if (result.RunId) {
            console.log('RunID:', result.RunId);
            return Number(result.RunId);
        } else {
            console.error('RunID not found in response');
            return null;
        }
    } catch (error) {
        console.error('Fetch error:', error.message);
        return null;
    }
};
```

Listing 13: Function to get RunId for Website (from `loginUtils.js`)



(a) RunID assignment process on ESP32 side



(b) RunID fetching process on website side

Figure 68: Processes of fetching and assigning RunID