

# Self-Balancing Robot

Course: ELEC 391 201 2024W2

Authors: Team B-17 (Tomaz Zlindra, Muntakim Rahman, Xianyao Li)

Instructors: Dr. Joseph Yan and Dr. Cristian Grecu

Submission Date: April 13th, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Requirements, Specifications and Constraints</b>	<b>3</b>
2.1	Functional Requirements . . . . .	3
2.2	Performance Requirements . . . . .	3
2.3	Constraints . . . . .	3
<b>3</b>	<b>Conceptual Design</b>	<b>3</b>
3.1	High-level Functional Diagram . . . . .	4
3.2	Architecture Diagram . . . . .	5
<b>4</b>	<b>Subsystem Design</b>	<b>5</b>
4.1	Structural Design . . . . .	5
4.1.1	Design Decisions . . . . .	5
4.1.2	Third Layer Design . . . . .	6
4.1.3	L-stand Design . . . . .	7
4.2	PID Modeling and Simulation . . . . .	7
4.2.1	Measuring Robot Parameters . . . . .	7
4.2.2	Transfer Function . . . . .	8
4.2.3	PID Parameter Calculation . . . . .	9
4.2.4	Simulink Simulation . . . . .	10
4.3	Arduino Nano 33 BLE Sense Rev2 . . . . .	11
4.3.1	Arduino IDE . . . . .	12
4.4	Angle Measurement . . . . .	12
4.4.1	IMU Sensors . . . . .	12
4.4.2	Complementary Filter . . . . .	12
4.5	Autonomous-Balancing . . . . .	13
4.6	Movement . . . . .	14
4.6.1	Forward/Backward Movement . . . . .	14
4.6.2	Left/Right Movement . . . . .	14
4.7	Bluetooth Control . . . . .	14
4.7.1	Robot Driver App . . . . .	15
4.8	STM32L051K8T6 Microcontroller . . . . .	16
4.9	HC-SR04 Ultrasonic Ranging Module . . . . .	17
4.10	TCS34725 Light-to-Digital Sensor . . . . .	18
4.11	RC522 Contactless Reader IC . . . . .	19
4.12	CEM-1302 Speaker/Buzzer . . . . .	20
4.13	Raspberry Pi Zero 2 W . . . . .	20
4.13.1	Camera App . . . . .	21
4.13.2	AWS Bucket . . . . .	22
<b>5</b>	<b>Verification and Validation</b>	<b>22</b>
5.1	Verification . . . . .	22
5.2	Validation . . . . .	22
<b>6</b>	<b>Conclusions and Future Work</b>	<b>22</b>
<b>7</b>	<b>References</b>	<b>22</b>
<b>8</b>	<b>Appendices</b>	<b>22</b>
8.1	Appendix: Budget . . . . .	23
8.2	Appendix: Technical Calculations and Simulations . . . . .	23
8.3	Appendix: Drawings, Schematics, and Blueprints . . . . .	23
8.4	Appendix: Datasheets / Product Specifications . . . . .	23
8.5	Appendix: Relevant Industry Standards and Regulations Used . . . . .	23
8.6	Appendix: Prototypes . . . . .	23

8.7	Appendix: Sample Code . . . . .	23
-----	---------------------------------	----

# 1 Introduction

- Description of the engineering problem or challenge
- Background context and any relevant history

## 2 Requirements, Specifications and Constraints

- It's helpful to list these in a table or enumerated, (e.g. R1, R2, etc.) so this can be treated as of checklist of requirements that must be tested as detailed in the Verification and Validation section.

### 2.1 Functional Requirements

The table below (1) lists all the functional requirements of the robot

ID	Requirement Description	Priority
FR1	The robot must balance on a flat surface by controlling the DC motors.	Core
FR2	The robot must maintain balance while driven externally on a flat surface.	Core
FR3	The robot must maintain balance while turning at a reasonable speed.	Core
FR4	The robot must move forward and backward under external control.	Core
FR5	The robot must be controlled via Bluetooth with four directional commands.	Core
FR6	The robot must maintain balance on a track with up to 15 degrees of incline.	Core
FR7	The distance sensor must detect obstacles within 5-50 cm and stop the robot within 20 cm.	Additional Feature
FR8	The color sensor must be able to detect at least red, green and blue colors	Additional Feature
FR9	The RFID module must authenticate an RFID tag within 1 second to prevent unauthorized operation.	Additional Feature
FR10	The Raspberry Pi Zero must process and transmit a camera feed at least 1 FPS.	Additional Feature

Table 1: Functional Requirements of the Self-Balancing Robot

### 2.2 Performance Requirements

Metrics for performance such as speed, efficiency, capacity, etc.

### 2.3 Constraints

(financial, temporal, material, environmental, standards and regulations to follow, etc.)

## 3 Conceptual Design

High level functional diagram/architecture of the overall system.

### 3.1 High-level Functional Diagram

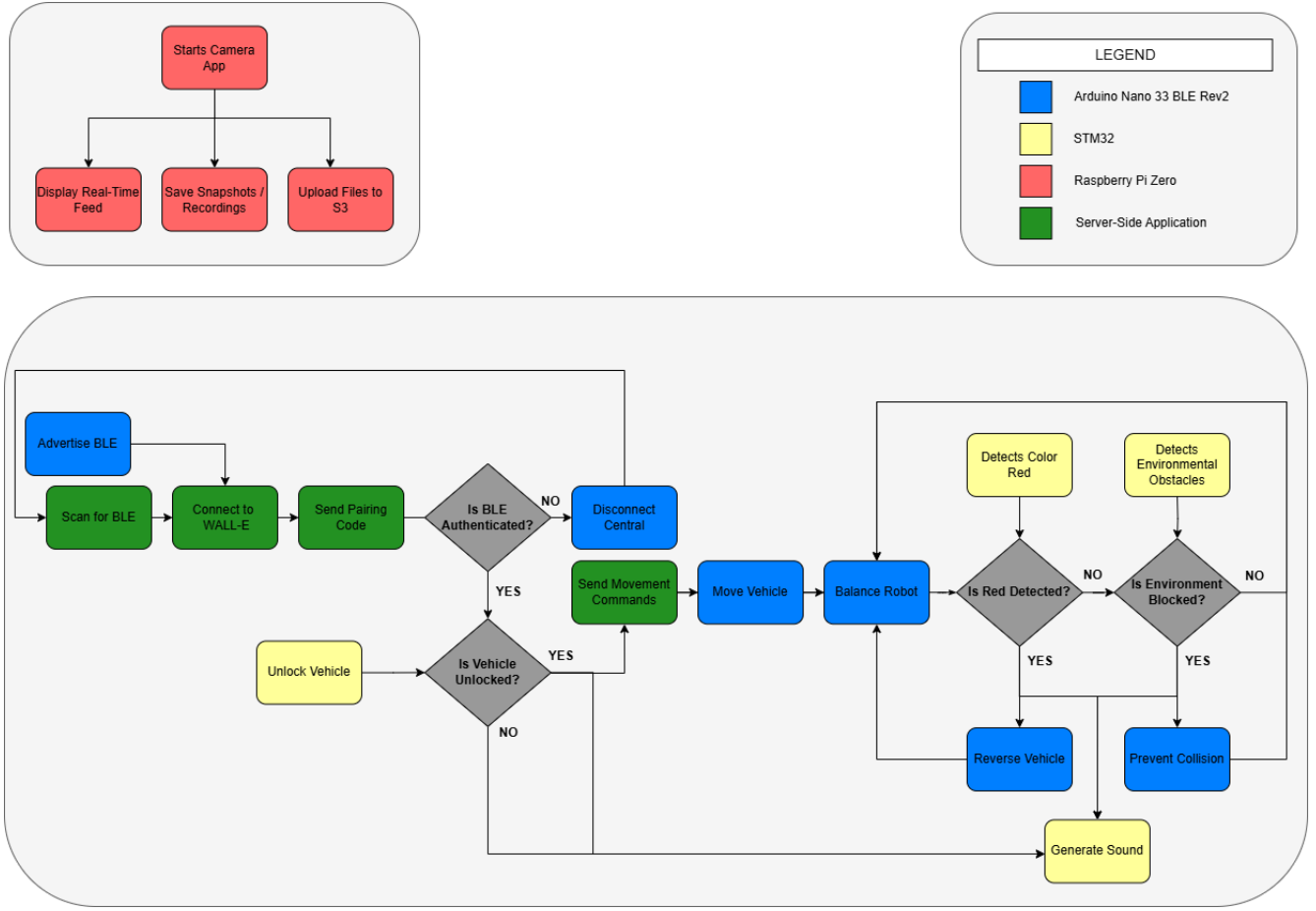


Figure 1: High-level Functional Diagram

The functional diagram in Figure (1) highlights our system-design for enforcing secure access of the robot with **RFID** and **Bluetooth Low Energy (BLE)**. We prevent users from balancing or moving the robot without the required authentication. Once successful, the firmware was programmed to prioritize safe operation of the robot, while accounting for environmental factors such as distance and color detection.

## 3.2 Architecture Diagram

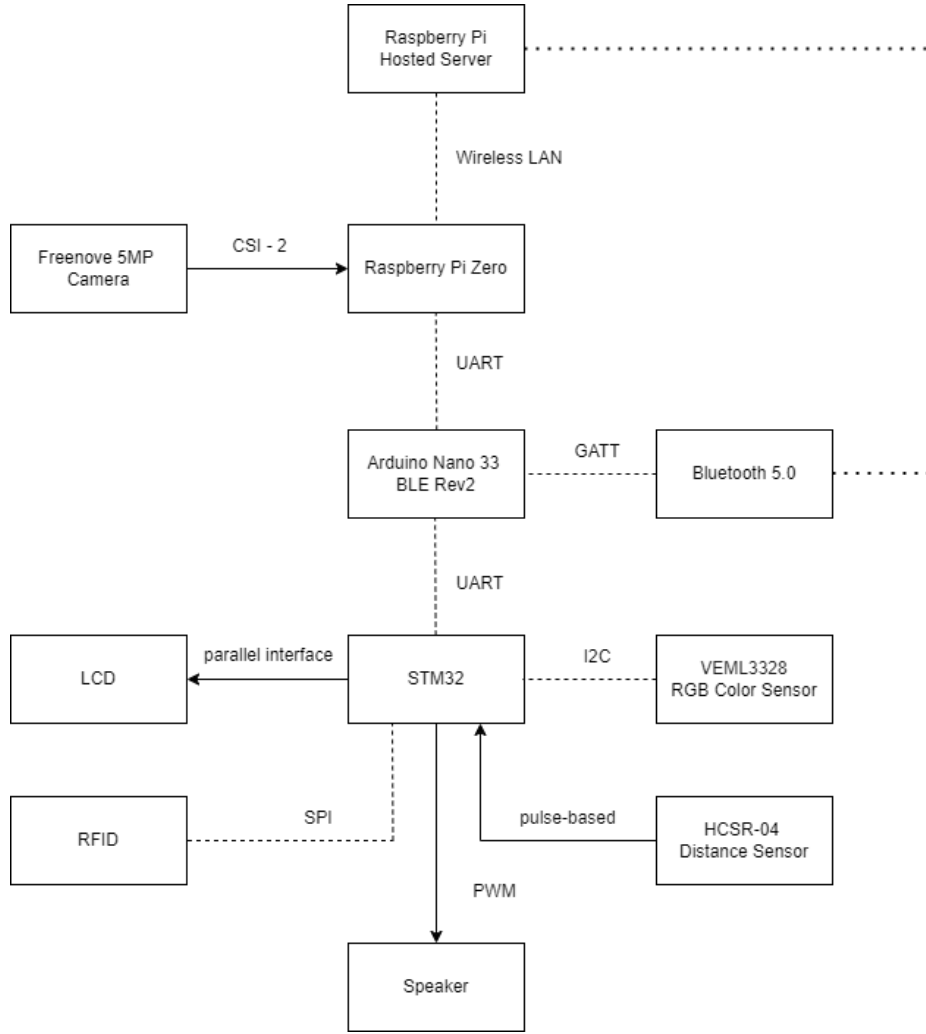


Figure 2: Architecture Diagram

This might need to be updated. We should also add some text to explain what is going on

## 4 Subsystem Design

Detailed description and schematics for key subsystems

Include electrical, mechanical, structural, or process-related subsystems as appropriate

### 4.1 Structural Design

#### 4.1.1 Design Decisions

Since the transfer function of the robot was derived under the assumption that the center of mass of the pendulum is perfectly centered, the structural design follows the principle of evenly distributing mass to ensure that the center of mass is as close as possible to the rotor axis.

Another design decision we made was to mount the third layer as high as possible. This choice addresses the fact that the plant is a nonlinear and inherently unstable system, which can only be effectively controlled within a region where it can be approximated as linear. To evaluate the impact of the third layer's height on system behavior, we simulated the plant's step response at different mounting heights:

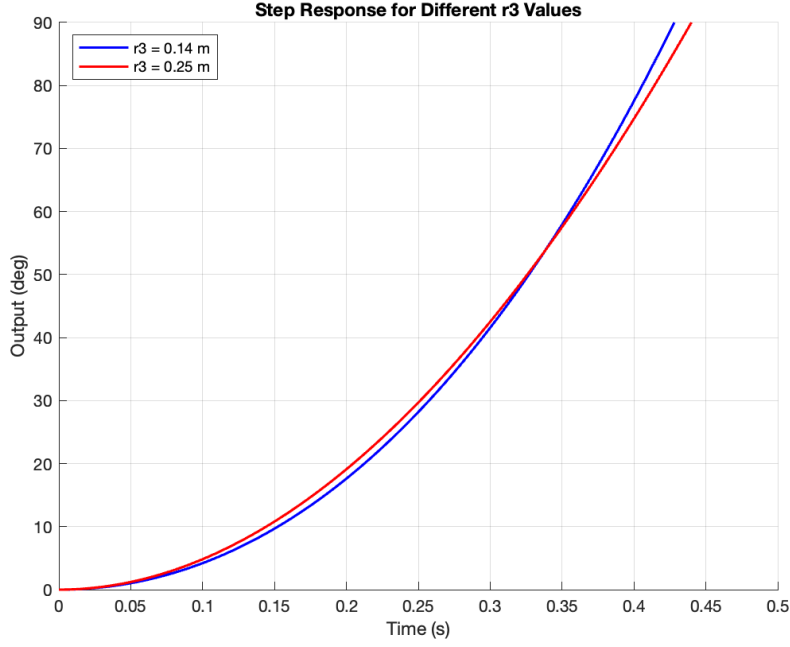


Figure 3: Step Response of the Plant with Different Height of the Third Layer

As shown in the figure, increasing the height of the third layer results in a step response that more closely resembles that of a linear system, making it easier to control.

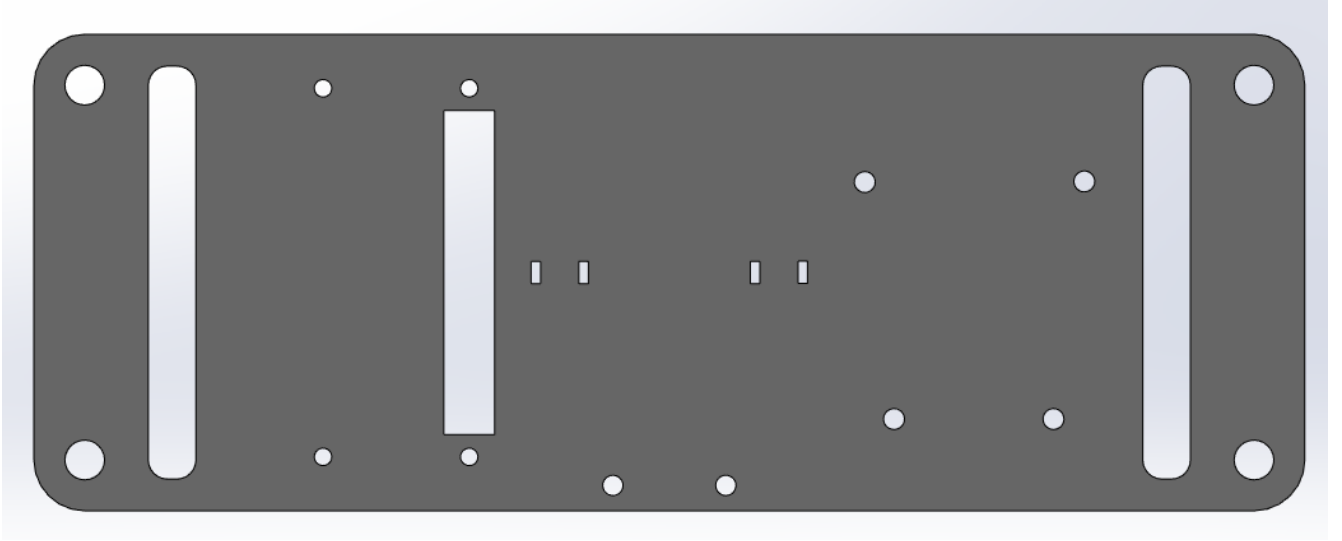


Figure 4: Design of the Third Layer

#### 4.1.2 Third Layer Design

On the third layer, we need to mount the Raspberry Pi, color sensor, RFID reader, and camera. The four mounting holes on the left side are for securing the Raspberry Pi, while the four holes on the right are for the RFID reader. These holes are positioned to ensure that both components are centered relative to the third layer once mounted. At the bottom, two mounting holes are used to attach the L-stand for the color sensor. Since we only have one color sensor, its placement causes a deviation in the center of mass. To compensate for this imbalance, the camera is zip-tied at the center, allowing its position to be adjusted to help realign the center of mass. Although not shown in the figure, Two distance sensors are also mounted on the second layer symmetrically.

### 4.1.3 L-stand Design

The design of the L-stand is shown below:

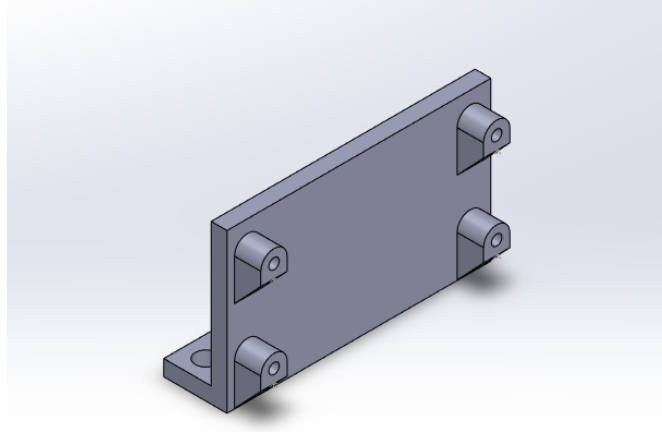


Figure 5: Design of the L-stand for Mounting the Distance Sensor

The L-stand includes four columns to provide clearance for the distance sensor, as there are components located on the back of the PCB. Ribs are added beneath each column to address potential issues during 3D printing. Since overhangs can lead to poor print quality or failed prints, the ribs provide structural support under the columns, eliminating overhangs and making the printing process more reliable and efficient.

The L-stand for the color sensor follows the same design principles with different dimension, thus is not shown here.

## 4.2 PID Modeling and Simulation

### 4.2.1 Measuring Robot Parameters

With the corrected transfer function, we need to find the values of the parameters in order to get the transfer function.

We start with the motor constants. For  $K_t$ , from the datasheet we know that the stall current is 5.5 A and the torque at 12 V is 85 kg·mm, or 0.83N·m, we can then get the value of  $K_t$  by dividing the torque by the stall current, which is 0.1516.

For  $K_e$ , from the datasheet we know that the free run speed of the rotor without the gearbox is 1047.19 rad/s, we can then get the value of  $K_e$  by dividing the load voltage 12 V by the free run speed of the rotor, which is 0.01146.

To obtain the moment of inertia of the pendulum  $J_p$ , we need to simplify the plant model. The robot consists of four rods and three layers. Assuming the center of mass is perfectly centered, the four rods can be approximated as a single equivalent rod located at the center of the three layers. Similarly, the three layers can be modeled as three point masses, which are further simplified into a single point mass located at their collective center of mass. The resulting simplified model consists of a single rod rotating about one end and a point mass positioned at the center of mass of the three layers.



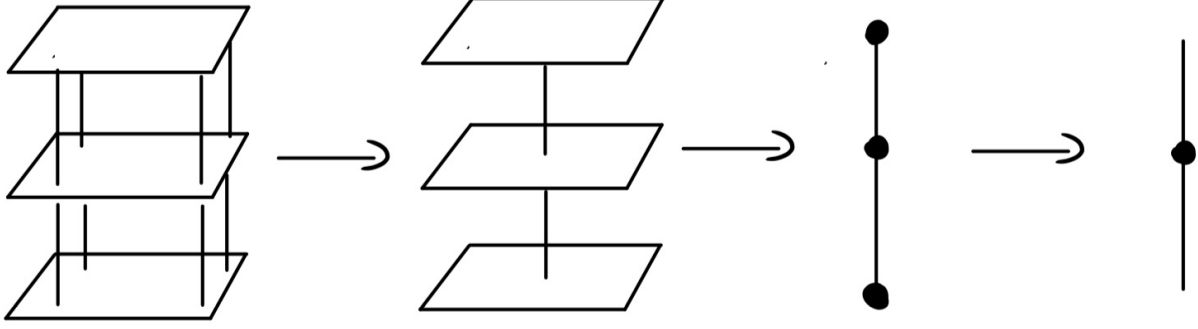


Figure 6: Simplification of the Plant Model

All of the other parameters can either be measured directly, or are negligible and have minimal impact on the overall transfer function.

#### 4.2.2 Transfer Function

Considering the difficulty of deriving the transfer function from first principles, and given the abundance of studies on the same topic, we chose to reference a published paper for guidance [1]. However, upon reviewing the paper, we identified errors in part of the derivation:

Assuming the correctness of

$$K_t \frac{V_s - K_e \dot{\theta}_m}{R_a} = J_m \ddot{\theta}_m + \tau_w + b \dot{\theta}_m \quad (1)$$

$$\tau_w = J_w \ddot{\theta}_w - F_{pH} r_w + m_w \ddot{\theta}_w r_w^2 \quad (2)$$

$$F_{pH} = m_p (r_w \ddot{\theta}_w + l_g \ddot{\theta}_p) \quad (3)$$

$$F_{pH} l_g = J_p \ddot{\theta}_p \quad (4)$$

$$\theta_m = \frac{\theta_w}{G_r} \quad (5)$$

Substitute (3) into (4) gives:

$$\begin{aligned} m_p (r_w \ddot{\theta}_w + l_g \ddot{\theta}_p) l_g &= J_p \ddot{\theta}_p \\ m_p l_g r_w \ddot{\theta}_w + m_p l_g^2 \ddot{\theta}_p &= J_p \ddot{\theta}_p \\ m_p l_g r_w \ddot{\theta}_w &= (J_p - m_p l_g^2) \ddot{\theta}_p \\ \ddot{\theta}_w &= \frac{J_p - m_p l_g^2}{m_p l_g r_w} \ddot{\theta}_p \\ \theta_w &= \frac{J_p - m_p l_g^2}{m_p l_g r_w} \theta_p \end{aligned}$$

So Eq. 4 in the paper is incorrect. The  $\tau_w$  in denominator should be  $r_w$ . The signs are also wrong. Substitute (3) into (2) gives:

$$\begin{aligned} \tau_w &= J_w \ddot{\theta}_w - m_p (r_w \ddot{\theta}_w + l_g \ddot{\theta}_p) r_w + m_w \ddot{\theta}_w r_w^2 \\ &= (J_w + (m_w - m_p) r_w^2) \ddot{\theta}_w - m_p l_g r_w \ddot{\theta}_p \end{aligned}$$

Substitute the above and (5) into (1) and expand the first fraction:

$$\frac{K_t}{R_a} V_s - \frac{K_t}{R_a} \frac{K_e}{G_r} \dot{\theta}_w - \frac{b}{G_r} \dot{\theta}_w - (J_w + (m_w - m_p) r_w^2) \ddot{\theta}_w + m_p l_g r_w \ddot{\theta}_p = \frac{J_m}{G_r} \ddot{\theta}_w$$

Move the angle terms to RHS and flip both sides:

$$\left(\frac{K_t}{R_a} \frac{K_e}{G_r} + \frac{b}{G_r}\right) \dot{\theta}_w + \left[\frac{J_m}{G_r} + J_w + (m_w - m_p)r_w^2\right] \ddot{\theta}_w - m_p l_g r_w \ddot{\theta}_p = \frac{K_t}{R_a} V_s$$

Substitute the expression of  $\theta_w$  in terms of  $\theta_p$ :

$$\begin{aligned} (J_p - m_p l_g^2) \left(\frac{K_t}{R_a} \frac{K_e}{G_r} + \frac{b}{G_r}\right) \dot{\theta}_p + \left[(J_p - m_p l_g^2) \left[\frac{J_m}{G_r} + J_w + (m_w - m_p)r_w^2\right] + (m_p l_g r_w)^2\right] \ddot{\theta}_p \\ = \frac{K_t}{R_a} (m_p l_g r_w) V_s \end{aligned}$$

Laplace transform:

$$\begin{aligned} \theta_p(s) \left[(J_p - m_p l_g^2) \left(\frac{K_t}{R_a} \frac{K_e}{G_r} + \frac{b}{G_r}\right) s + \left[(J_p - m_p l_g^2) \left[\frac{J_m}{G_r} + J_w + (m_w - m_p)r_w^2\right] + (m_p l_g r_w)^2\right] s^2\right] \\ = \frac{K_t}{R_a} (m_p l_g r_w) V_s(s) \end{aligned}$$

Hence the final transfer function:

$$\frac{\theta_p}{V_s}(s) = \frac{\frac{K_t}{R_a} (m_p l_g r_w)}{\left[(J_p - m_p l_g^2) \left[\frac{J_m}{G_r} + J_w + (m_w - m_p)r_w^2\right] + (m_p l_g r_w)^2\right] s^2 + (J_p - m_p l_g^2) \left(\frac{K_t}{R_a} \frac{K_e}{G_r} + \frac{b}{G_r}\right) s}$$

The definition of the parameters can be found in the paper.

Plug in the parameters we acquired, the resulting open loop transfer function is:

$$\frac{\theta_p}{V_s}(s) = \frac{977.5}{s^2 + 0.3439s}$$

Notice that the moment of inertia of the wheel and the rotor ( $J_m$  &  $J_r$ ) and friction  $b$  are approximated to zero, This is because their magnitudes are negligible compared to other parameters in the system, thus they have minimal impact on the overall transfer function.

#### 4.2.3 PID Parameter Calculation

From the previously derived transfer function, we observe that the two open-loop poles are located at 0 and -0.3439, indicating that the system is unstable. To stabilize the system, a feedback control loop must be introduced to shift the poles of the closed-loop transfer function to stable locations. To determine the PID parameters, we can apply the pole placement method:

Assuming that we want the resulting poles to be placed at  $P_A$  and  $P_B$ , the denominator of the resulting transfer function is:

$$(s - P_A)(s - P_B)$$

This can be written explicitly as:

$$s^2 - (P_A + P_B)s + P_A P_B$$

With a PD controller, the characteristic equation is:

$$s^2 + (K \cdot K_d - (P_1 + P_2))s + P_1 P_2 + K K_p$$

Where  $P_1$  and  $P_2$  are poles of the open loop transfer function,  $K$  is the DC gain of the transfer function, which is 977.5.

We can relate the above two equations and the following equations can be obtained:

$$\begin{aligned} P_A P_B &= P_1 P_2 + K K_p \\ -(P_A + P_B) &= K K_d - (P_1 + P_2) \end{aligned}$$

The resulting PD gains will make the close loop transfer function be critically damped and stable. To determine the desired pole locations, we first need to select an appropriate time constant  $N$ . By measuring the runtime of the main control loop, we found the worst-case execution frequency to be 25 Hz. According to the Nyquist theorem, the sampling rate must be at least twice the highest frequency component to accurately capture the system dynamics. Therefore, the maximum allowable control bandwidth is 12.5 Hz, corresponding to a minimum time constant of approximately 0.08. To ensure stability and responsiveness while staying within this limit, we chose a time constant of 1/4 seconds, which places the desired pole at  $-8\pi$  and the resulting  $K_p$  and  $K_d$  are 0.6462 and 0.0511, respectively.

The I term is added after to address the drifting issue we encountered during the testing. The value of  $K_i$  was found by trial and error, and the final value of  $K_i$  is 11.5.

#### 4.2.4 Simulink Simulation

Simulink is used to validate the design of the PID controller. The model is shown below:

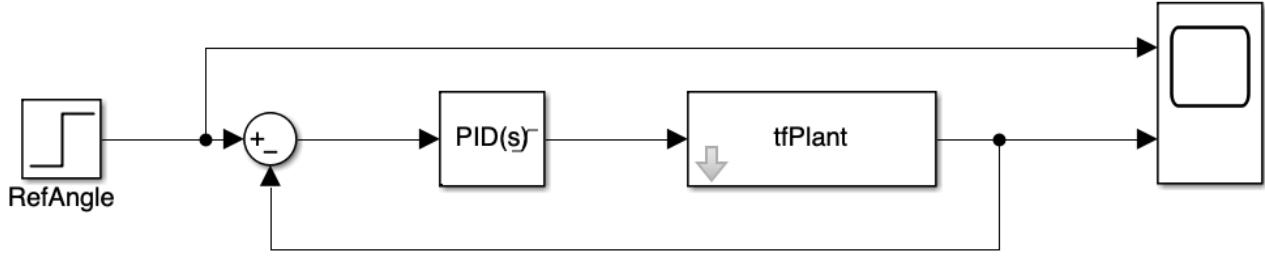


Figure 7: Simulink Model for the Bot

The step responses of the system under filter coefficient  $N = 50$  and  $N = 25$  were simulated to evaluate the performance of the robot under best- and worst-case scenarios:

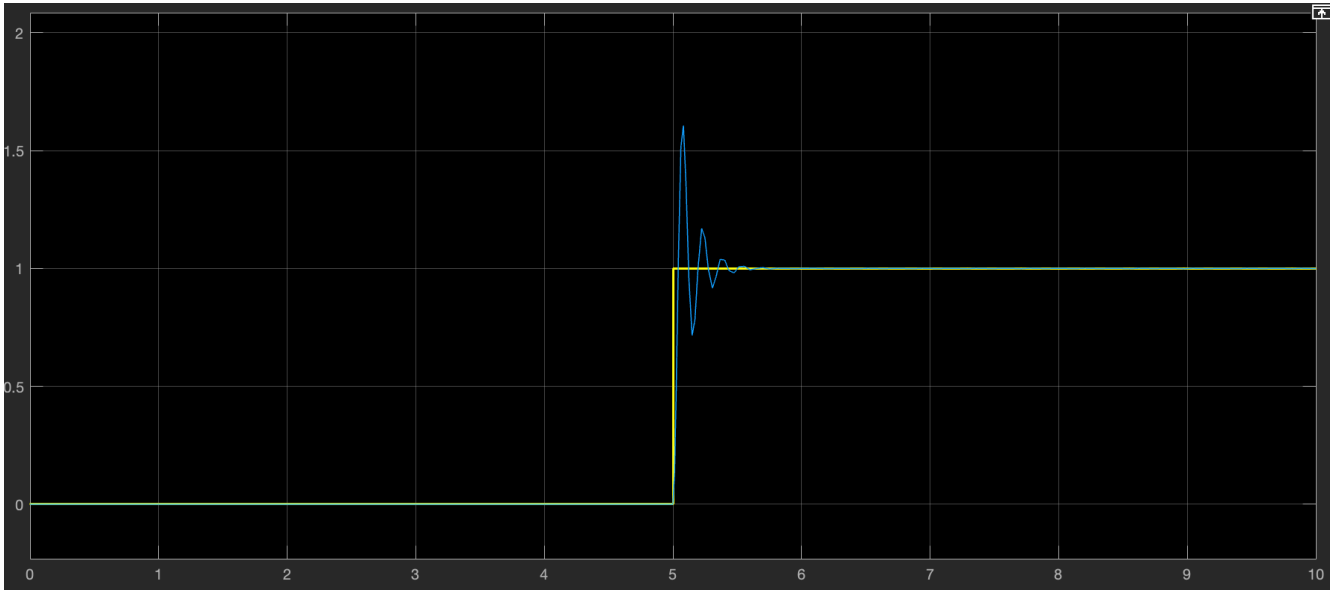


Figure 8: Step Response of the System with  $N = 25$

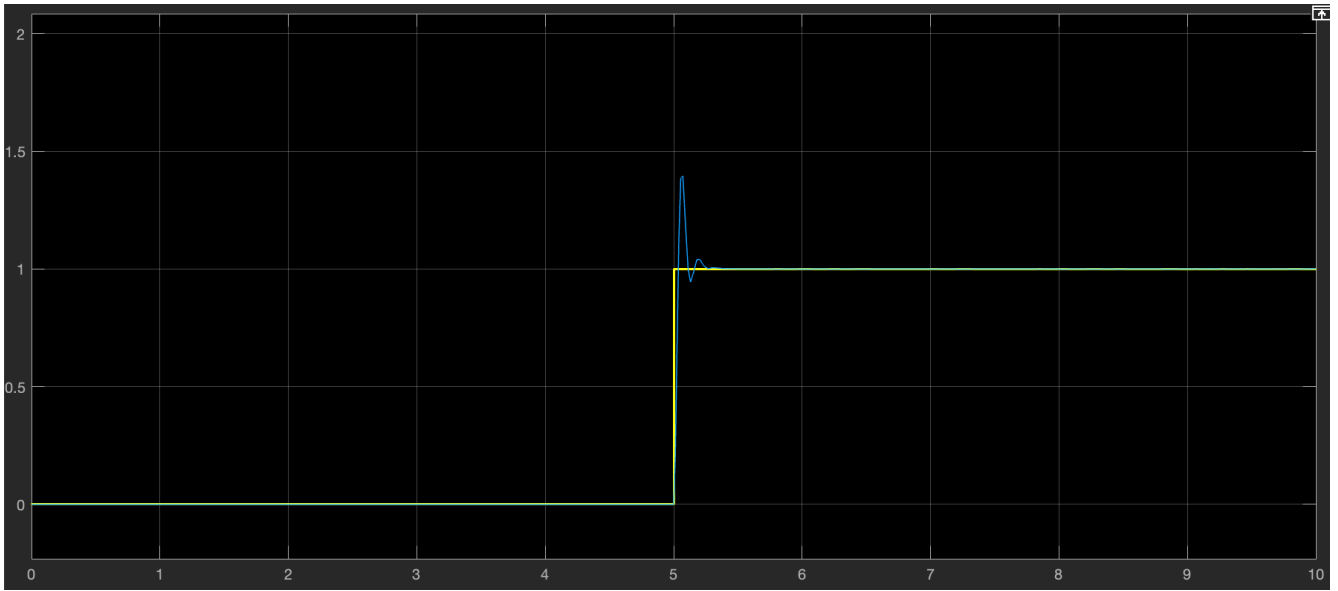


Figure 9: Step Response of the System with  $N = 50$

We can see that for both cases the system is stable, with different degree of overshoot. This implies that the design of the PID controller is effective.

### 4.3 Arduino Nano 33 BLE Sense Rev2

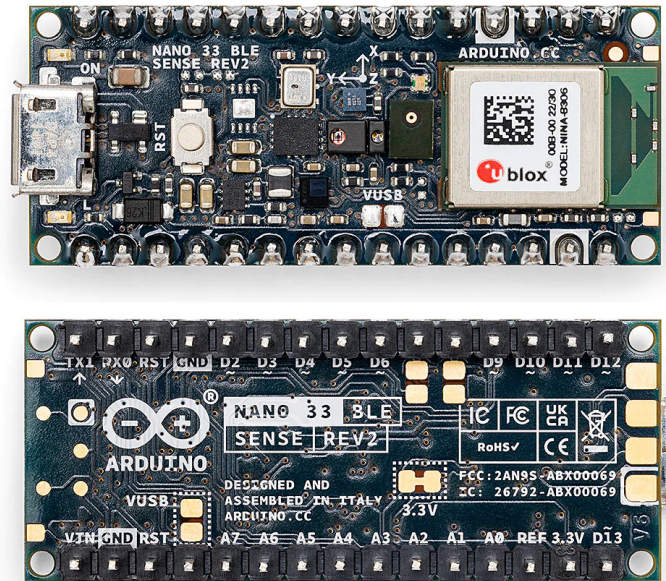


Figure 10: Arduino Nano 33 BLE Sense Rev2

The **Arduino Nano 33 BLE Sense Rev2** is a compact and powerful microcontroller board designed for low-power applications. It features the Nordic **nRF52840** chipset, which supports **BLE** communication, includes a range of built-in sensors, including a 9-axis IMU (accelerometer, gyroscope, and magnetometer). These integrated features make it well-suited for IoT, sensor-based applications, and portable devices.

This made it the ideal choice for our self-balancing robot project and was responsible for many of the core features. It integrated key core components in this project, particularly the IMU and the **BLE** module.

#### 4.3.1 Arduino IDE

The **Arduino IDE** was integral in quickly testing new firmware changes to the board with built-in compilation and flashing tools. The extensive library support significantly streamlined the development process, reducing the need for bare metal programming and allowing us to focus on the functionality.

### 4.4 Angle Measurement

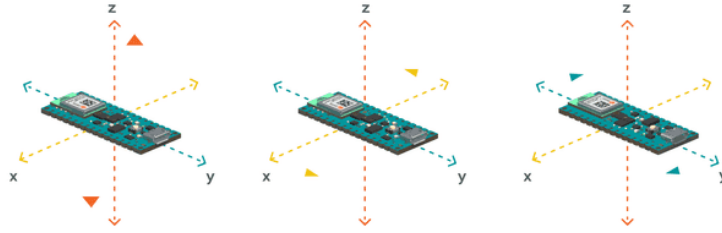


Figure 11: IMU

The **Inertial Measurement Unit (IMU)** was a critical component of the microcontroller for measuring velocity, orientation, and gravitational forces. This enabled the tracking of motion and orientation of the **Arduino** in three-dimensional space.

#### 4.4.1 IMU Sensors

We used the accelerometer to measure linear acceleration and the gyroscope to monitor rotational movement. Both of these sensors had advantages and disadvantages, as shown in Table 2.

By combining them, we ensured reliable and accurate angle measurements for optimized balancing in the control feedback loop.

#### 4.4.2 Complementary Filter

The solution to combine the strengths of both sensors was to use a complementary filter, which combines the accelerometer and gyroscope readings with their respective normalized weights.

Since the gyroscope provided a rough estimate of the angle change, this was essential for detecting quick movements.

The formula used for the complementary angle is:

$$\theta_n = k(\theta_{n-1} + \theta_{g,n}) + (1 - k)\theta_{a,n},$$

where:

- $\theta_n$  is the current complementary angle,

- $\theta_{n-1}$  is the previous complementary angle,
- $\theta_{g,n}$  is the current gyroscope angle,
- $\theta_{a,n}$  is the current accelerometer angle,
- $k$  is the normalized weight.

$\therefore$  From extensive testing, a value of  $k = 0.95$  was selected.

*Lower values of  $k$  caused the noise from the accelerometer to become significant, leading to large oscillations and loss of balance. Similarly, increasing  $k$  was avoided. The gyroscope bias would accumulate quickly (i.e., without a reference point) and cause the robot to lose balance.*

The code snippet for this exact implementation is demonstrated in Listing 1.

## 4.5 Autonomous-Balancing

The most challenging part of this project was to implement the PID controller to balance the robot. Since we had closed-loop control, with the measured angle acting as feedback, it was critical that we were able to act upon new measurements when determining the control signal.

Initially, we attempted to implement these **Interrupt Service Routine (ISR)** to meet a designed control frequency of 20 Hz. However, this proved to accumulate complications in firmware logic and was not able to provide the expected results.

We ultimately designed a control loop, run in the main program. This followed the sequence shown in Figure 12.

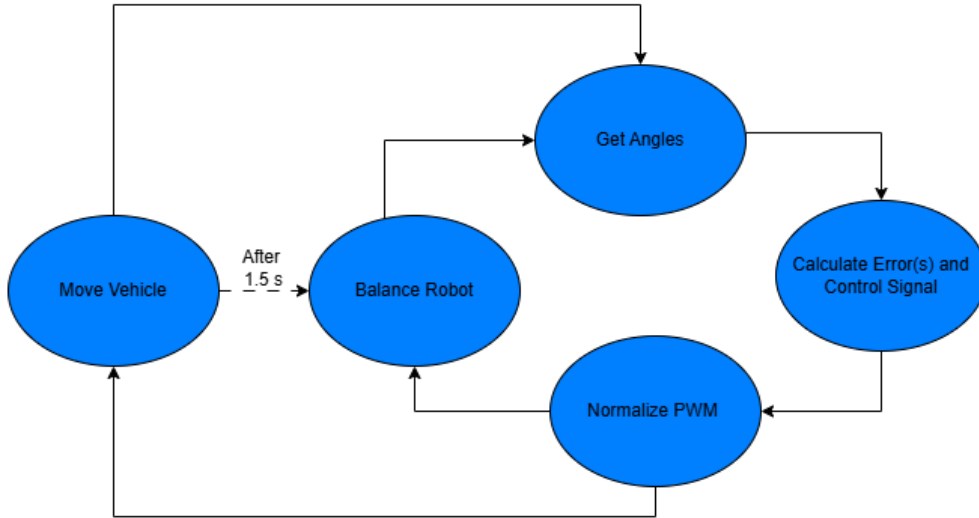


Figure 12: Control Loop Diagram

This ensured that we were able to inject **PWM** signals to both motors, without the constraints typical of an **ISR**. At the same time, this ensured that the robot was acting upon the most recent measurements.

The sign of the control signal determined whether to move the robot forward / backward to maintain balance.

## 4.6 Movement

The robot was designed to move in four directions: **FORWARD, BACKWARD, LEFT, RIGHT**. As shown in Figure 12, this was designed to be integrated into the control loop.

Balancing was the highest priority of the system at any given time. To move the robot, we injected noise into the system for a short period of time (i.e. 1.5 seconds). After this, we returned to idle balancing to prevent the system from reaching a point of instability.

### 4.6.1 Forward/Backward Movement

The logic for moving the robot forward and backward was very simple. It was a matter of adjusting the setpoint angle on either side of the "zero point". This induced movement in the intended direction (without sacrificing balance).

### 4.6.2 Left/Right Movement

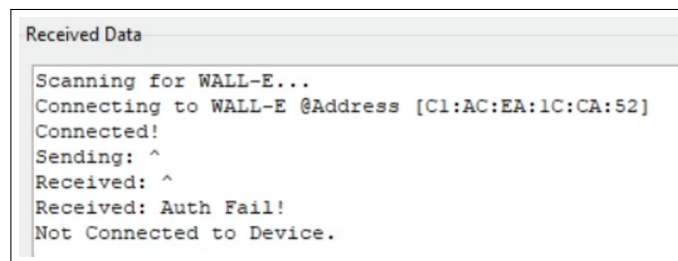
The logic for moving the robot left and right was slightly more complicated and involved software logic to interchangeable turn and balance the robot. This can be summarized as follows:

- Every 3 iterations of control loop : balance robot
- If error angle > allowable threshold (i.e. 0.8 degrees): balance robot
- Otherwise:
  - To turn left: move left motor *CW* and right motor *CCW*, with **PWM** adjusted by scale factor 0.6
  - To turn right: move left motor *CCW* and right motor *CW*, with **PWM** adjusted by scale factor 0.5

## 4.7 Bluetooth Control

The robot was controlled via the **Arduino's Bluetooth Low Energy (BLE)** capabilities. The firmware was developed such that new connections need to be authorized by successfully transmitting the correct pairing code. This follows the workflow shown in Figure 1.

In order to send movement commands, the connection and authentication process must be successfully complete to ensure secure access. A key part of this process was to provide the expected pairing code to the microcontroller.



```
Received Data
Scanning for WALL-E...
Connecting to WALL-E @Address [C1:AC:EA:1C:CA:52]
Connected!
Sending: ^
Received: ^
Received: Auth Fail!
Not Connected to Device.
```

Figure 13: Unauthorized Bluetooth Connection

If unsuccessful, the robot would immediately disconnect and await an authorized user as shown in Figure 13.

The firmware is also designed such that movement commands are placed on a lower priority. Since **BLE** can be computationally expensive, we polled for new commands every 100 ms in the main loop. This was done to ensure that the control functionality is placed at highest priority, and we can account for the infrequent nature of user driven commands.

#### 4.7.1 Robot Driver App

We developed a **Flask** application, which used **GET/POST** request **HTTP** methods to asynchronously request device information from the **HTML DOM** elements and send commands to the **Arduino** microcontroller.

This application was run from our computer and temporarily hosted to the Internet using the **ngrok** tool. This enabled us to access the robot via public URL from other devices (including smartphones).

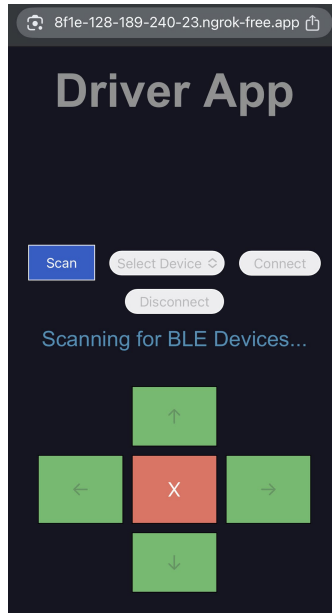


Figure 14: Robot Driver App on iPhone 14

The **Flask** application had the expected pairing code programmed in the source code such that users can easily connect to and control the robot. We programmed the **DRIVE**, **LEFT**, **RIGHT**, **BACK** commands to correspond to specific bytes to be transmitted via **BLE**.

This is shown in the code snippet in Listing 6 in the Appendix.



## 4.8 STM32L051K8T6 Microcontroller

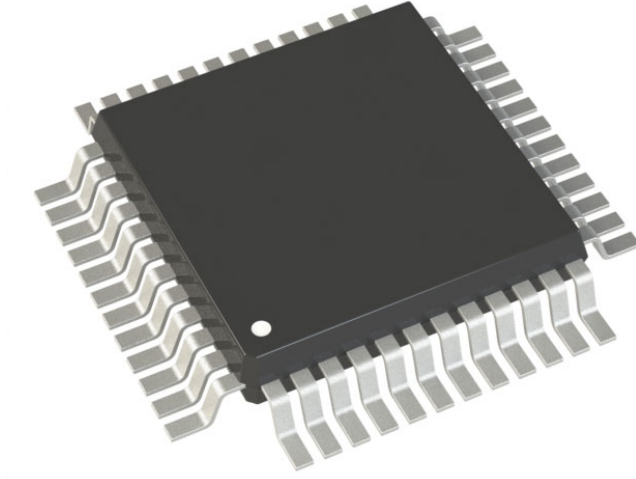


Figure 15: STM32L051K8T6 Microcontroller

The STM32L051K8T6 is a low-power microcontroller from STMicroelectronics, part of the STM32L0 series. It features an ARM Cortex-M0+ core running at up to 32 MHz, with 64 KB of flash memory and 8 KB of SRAM. The microcontroller offers a variety of peripherals, including I<sup>2</sup>C, SPI, UART, ADC, and GPIO, along with multiple low-power modes. For our project, this microcontroller was used to interface with the HC-SR04 (section 4.9), TCS34725 (section 4.10), RC522 (section 4.11) and the CEM-1202 (section 4.12) modules.

Due to the limited number of peripherals on the STM32L051K8T6 microcontroller comprising 2 USART ports, 1 I<sup>2</sup>C, and 1 SPI interface, along with only 27 available I/O pins—we were required to carefully allocate communication protocols to each sensor (I<sup>2</sup>C for the color sensor and SPI for the RFID sensor). We opted not to use UART to maintain simplicity and efficiency. Out of the 27 I/O pins, we utilized approximately 20. It is worth noting that although USART is enabled in the configuration (as shown in Figure 16), it was never used in the implementation.

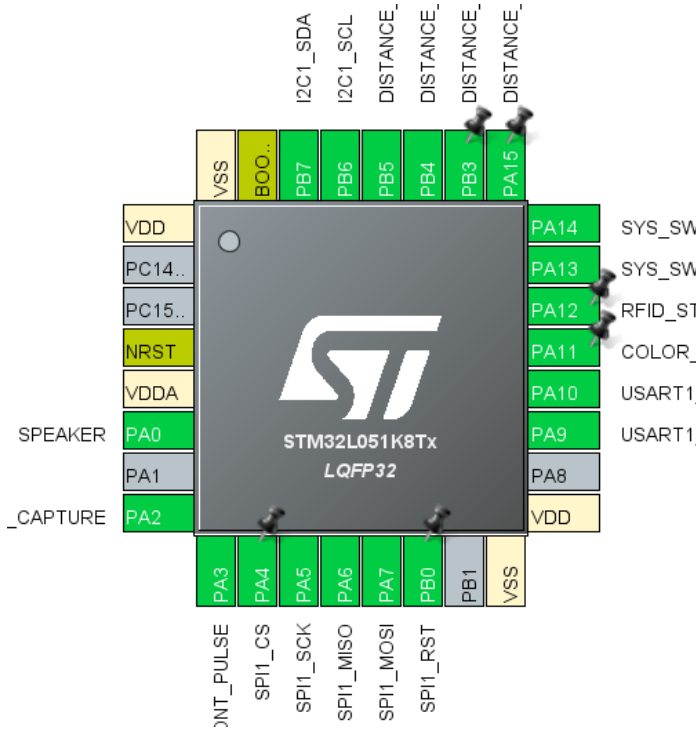


Figure 16: Zoomed In Pinout

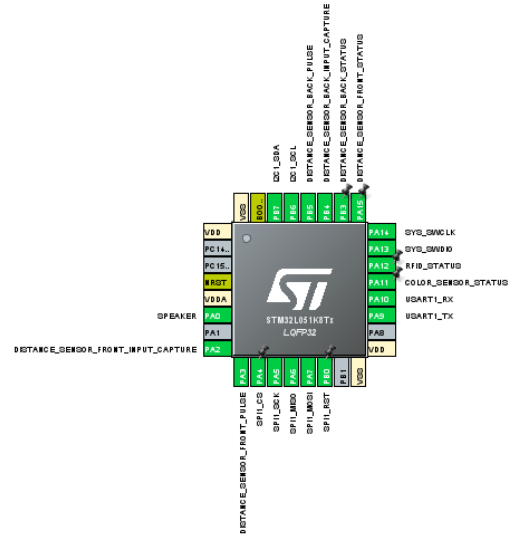


Figure 17: Zoomed Out Pinout

To streamline the firmware development process for the STM32, we utilized the STM32Cube IDE, provided by STMicroelectronics. This development environment facilitated faster progress by offering built-in tools and libraries, allowing us to focus on application logic rather than dealing with bare-metal programming.

#### 4.9 HC-SR04 Ultrasonic Ranging Module

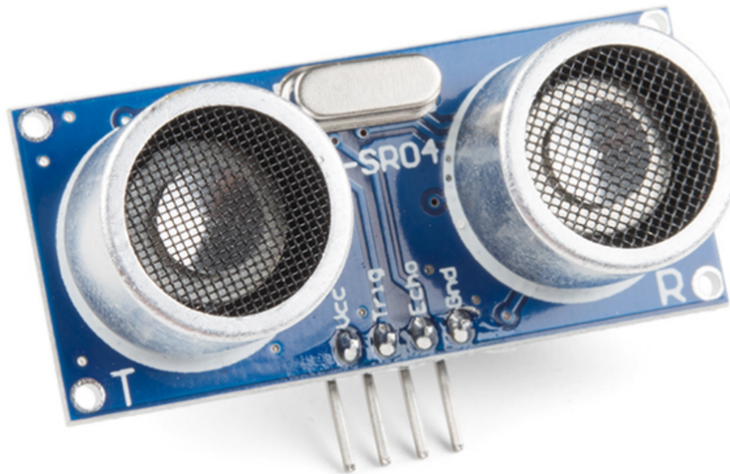


Figure 18: HC-SR04 Ultrasonic Ranging Module

The **HC-SR04** is an ultrasonic distance measurement module commonly used in embedded systems for sensing proximity to objects. It operates by emitting an ultrasonic pulse from its transmitter and measuring the time it takes for the pulse to reflect off a nearby surface and return to the receiver. The module communicates via a simple TTL interface, requiring a 'trigger' signal to start the measurement and an 'echo' signal to indicate when

the reflected pulse is received. By calculating the time interval between the trigger and echo signals, the distance to the object can be determined.

Firmware for interfacing with this module was developed on the *STM32L051* microcontroller, chosen for its ease of integration via HAL libraries and to offload processing from the Arduino Nano. Since the STM32 also handled other peripherals, efficient use of its timer channels with interrupts was essential to avoid overloading the main control loop.

The timer was configured with a prescaler of 31 (yielding a 1MHz timer frequency) to provide microsecond-level precision. One channel was set to PWM mode with a 10-cycle pulse ( $10\mu\text{s}$ ) to drive the Trigger pin. Another channel was configured in input capture mode, connected to the Echo pin. This setup allowed interrupts to be triggered on both rising and falling edges of the Echo signal, enabling precise capture of timer values at each transition. By handling these events in interrupt service routines, the CPU remained free to execute other tasks concurrently with the distance measurement process.

A code sample for the firmware implementation of input capture interrupt are shown in Listing 2 in the Appendix. More information about input capture mode and its implementation can be found on this site, [here](#).

#### 4.10 TCS34725 Light-to-Digital Sensor

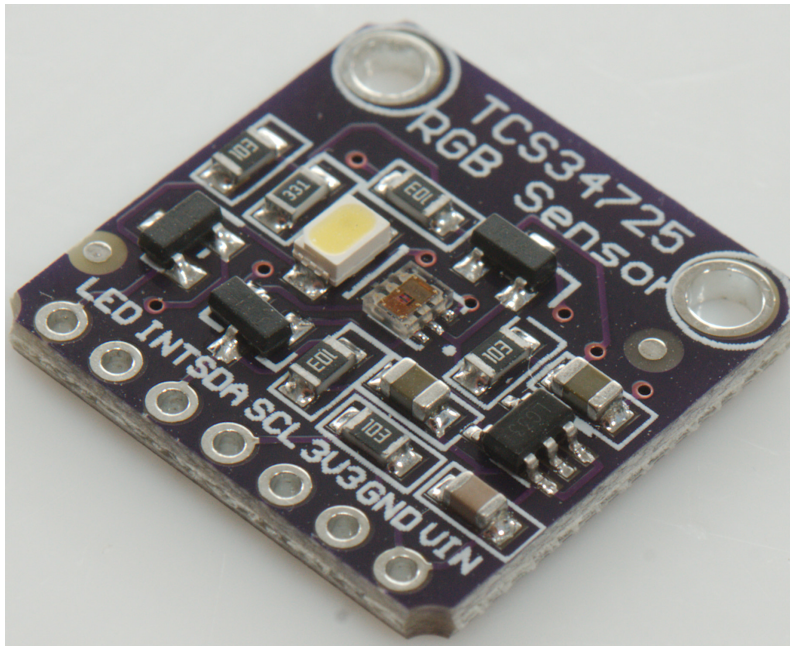


Figure 19: TCS34725 Light-to-Digital Sensor

The **TCS34725** Color Sensor Module was responsible for detecting the color composition of a surface by measuring the intensity of red, green, blue, and clear light. It communicates via an I<sup>2</sup>C interface and features an integrated IR filter to improve color accuracy under varying lighting conditions. When illuminated, typically by its onboard white LED, the sensor captures reflected light and provides digital values corresponding to each color channel. These values can be used to determine the overall color or detect specific color patterns in an environment.

This module was implemented on the *STM32L051* microcontroller instead of the Arduino Nano to reduce unnecessary clock cycle usage associated with I<sup>2</sup>C communication. This decision allowed more processing time to be allocated to tasks such as maintaining the balance of the robot and BLE communication. The STM32 HAL libraries simplify I<sup>2</sup>C communication, as demonstrated in Listing 3. Although we could've used DMA or interrupts for this module, we decided to not make it unnecessarily complicated and just left it in the main loop, as is.

The `ColorSensor_Handle` function is called within the main control loop. It reads the values from each color channel register (16 bits) and determines the most dominant color. If the red channel is the most prominent, the speaker is triggered to notify the user of the sensor's detection.

#### 4.11 RC522 Contactless Reader IC

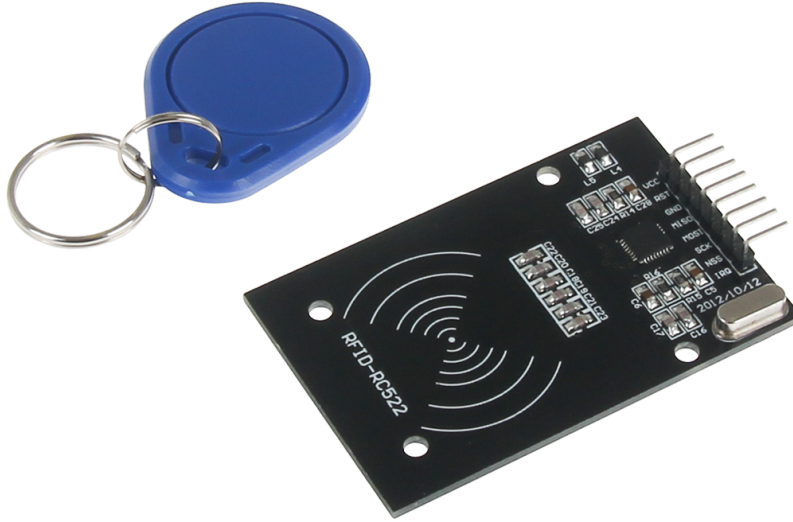


Figure 20: RC522 Contactless Reader IC

The **RC522** RFID Module was responsible for detecting and reading passive RFID tags using radio frequency communication at 13.56MHz. It operates over an SPI interface, allowing for fast and efficient data exchange with a microcontroller. When a tag enters the RF field generated by the module's antenna, the RC522 initiates a handshake protocol and reads the tag's unique identifier (UID) stored in its internal memory. This UID can then be used for identification or authentication purposes in embedded systems.

This module was also interfaced with the *STM32L051* microcontroller, primarily due to its ease of use and to offload processing responsibilities from the Arduino Nano. Given the large number of configuration registers required by the RC522 chip, an existing STM32 implementation was used as a reference, and its library was integrated into our firmware. The remaining logic specific to our application is shown in Listing 4 in the Appendix.

## 4.12 CEM-1302 Speaker/Buzzer



Figure 21: CEM-1302 Speaker/Buzzer

The **CEM-1302** speaker is a compact piezoelectric buzzer designed to produce audible tones. It operates by generating sound through the vibration of a diaphragm when an AC signal is applied. This module is commonly used in embedded systems to provide simple, high-pitched alerts or notifications.

The speaker was driven by the *STM32L051* microcontroller, utilizing timers in PWM mode. Given that the resonant frequency of the buzzer is approximately 2.048 kHz, we configured the timer with a prescaler of 31 (32 - 1) to achieve an overflow frequency of 1 MHz. With this setup, we set the capture compare value to 488 and the pulse width to 244, resulting in a PWM signal that generates a frequency close to 2.048 kHz with a 50% duty cycle.

This buzzer was used to provide feedback for the following events:

- RFID: Indicating successful and unsuccessful RFID card scans
- Ultrasonic Sensor: Alerting when the distance between the robot and an object is too small
- Color Sensor: Signaling when the color sensor detects a predominantly red color

When any of these events occurred, the speaker would either beep, or stay on, depending on the scenario. Code for the setup can be seen in Listing 5. GPIO pins for these events were used to communicate with the Arduino Nano rather than UART for communication, as its higher overhead and slow speed was deemed unfit for the task at hand.

## 4.13 Raspberry Pi Zero 2 W

The **Raspberry Pi Zero 2 W** is an affordable single-board computer designed for embedded systems and IoT applications. We leveraged its camera connector (CSI-2 interface) to interface with a **Freenove 5MP Camera**, as well as its Wi-Fi capabilities to wirelessly access it the **Raspberry Pi Connect** software.

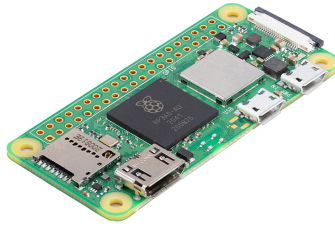


Figure 22: Raspberry Pi Zero 2 W

#### 4.13.1 Camera App

We developed a **Python** application to display the live camera feed in a **Tkinter** GUI window, allowing the user to observe and record the operation of the robot. The user interface provided the ability to also take snapshots and store these in the **SD Card** storage.

The live feed was largely a result of scheduling an update function with **Tkinter**'s `after(...)` method, which allowed us to sample camera frames at a rate of 20 FPS.

Since the **Raspberry Pi Zero 2 W** had power and performance constraints (i.e. 512 MB of SDRAM), we ensured that the application was optimized for performance and iterated software versioning to minimize resource consumption. The camera was configured to capture frames at a resolution of 480 pixels to account for this.

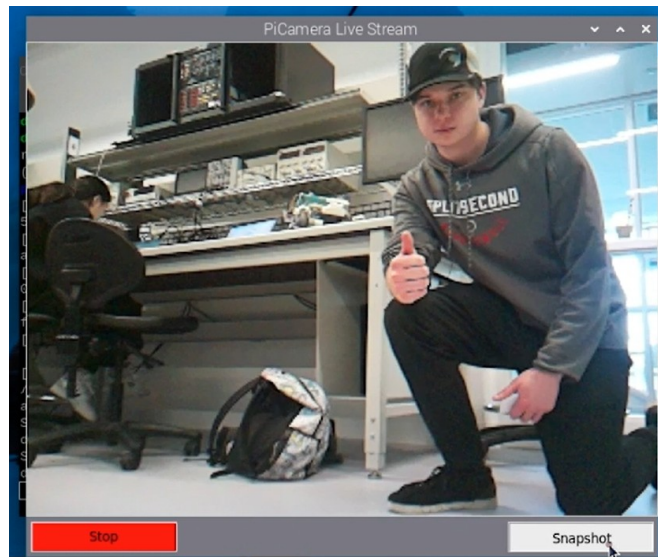


Figure 23: PiCamera Application

The class and methods for the camera display are shown in Listing 7 in the Appendix.

Early on in development, we switched from the **Kivy** framework. This provided smoother animations, but rendering was hardware accelerated. We were unconfident with the **Raspberry Pi Zero 2 W**'s ability to provide adequate performance with this, and ported the application to the current implementation.



### 4.13.2 AWS Bucket

Another constraint we considered during implementation was the inability to connect to the **Raspberry Pi** without powering on the robot. Having to access recordings and snapshots would be quite cumbersome for the user.

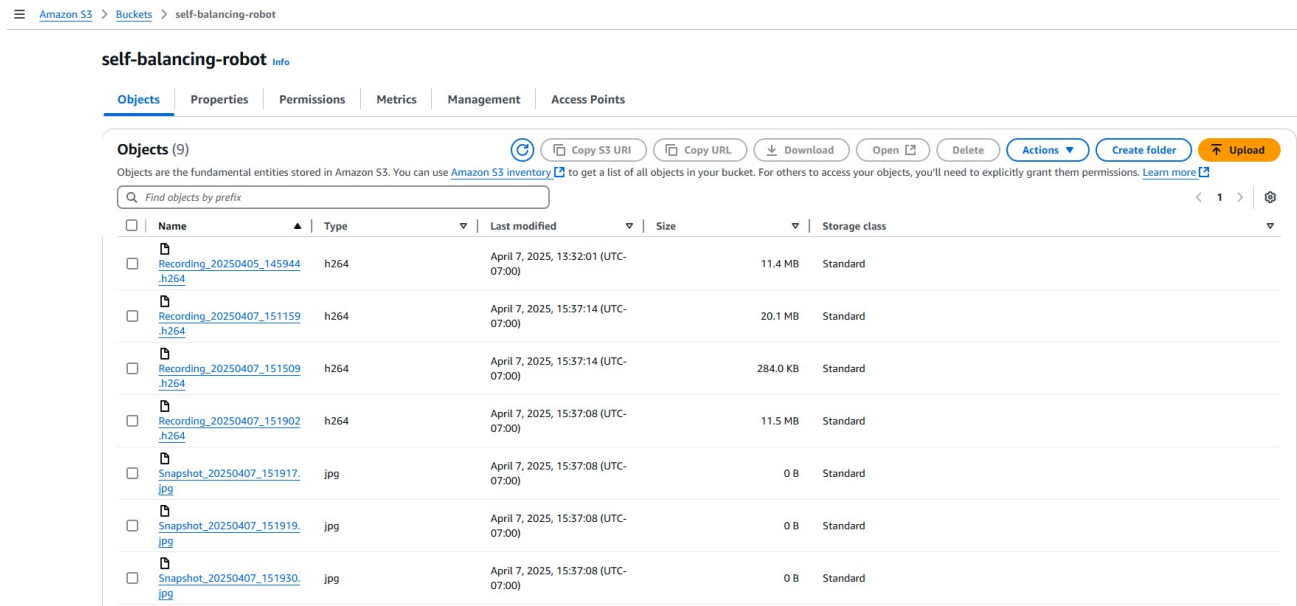


Figure 24: AWS S3 Bucket

We created an **AWS S3 Bucket** and uploaded the camera recordings and snapshots during operation. The **Boto** package enabled us to implement this in our **Python** application by leveraging the **AWS SDK**. We were able to download these files from the **AWS S3 Bucket** to our local machine for offline access.

## 5 Verification and Validation

### 5.1 Verification

testing methods and results to demonstrate that the product meets the specified requirements.

### 5.2 Validation

: Testing methods and results to demonstrate that the product actually will meet the high-level goals.

## 6 Conclusions and Future Work

Summary of the outcomes achieved/learned Recommendations for next steps or further improvements

## 7 References

Cite any academic papers, industry standards, or prior work consulted in the project. Use consistent reference formatting (e.g. <https://pitt.libguides.com/citationhelp/ieee> )

## 8 Appendices

Include sections that do not fit well in the main body of the report but help to show the development progress and justify your decisions. These might include (but may not be restricted to) the following:

## 8.1 Appendix: Budget

(list all expenditures including any cost overrun)

## 8.2 Appendix: Technical Calculations and Simulations

Table 2: IMU Sensors: Pros and Cons

Sensor	Pros	Cons
Accelerometer	<ul style="list-style-type: none"><li>• accurate gravitational readings</li><li>• zero-mean noise</li></ul>	<ul style="list-style-type: none"><li>• high noise variance (especially with motor vibrations)</li></ul>
Gyroscope	<ul style="list-style-type: none"><li>• lower noise over short time periods</li></ul>	<ul style="list-style-type: none"><li>• accumulates bias over time</li></ul>

## 8.3 Appendix: Drawings, Schematics, and Blueprints

## 8.4 Appendix: Datasheets / Product Specifications

Supporting Datasheets and Technical Documentation for Core Features (does not include battery related components, mechanical components, or generic electrical components):

1. Arduino Nano 33 IoT (ABX00069) Datasheet
2. Pololu 37D Metal Gearmotor Specifications
3. Texas Instruments DRV8833 Dual H-Bridge Motor Driver Datasheet

Supporting Datasheets and Technical Documentation for Additional Features:

1. HC-SR04 Ultrasonic Distance Sensor
2. TCS34725 Color Sensor
3. Raspberry Pi Zero 2 W
4. CN0090 Accelerometer Board
5. L293D Motor Driver IC
6. Freenove 5MP Camera (FNK0056)
7. CEM-1302 Speaker/Buzzer

## 8.5 Appendix: Relevant Industry Standards and Regulations Used

## 8.6 Appendix: Prototypes

## 8.7 Appendix: Sample Code

```
1  ANGLES Angles = {0, 0, 0}; // Accelerometer, Gyroscope, Complementary
2  void getAngles(ANGLES &Angles) {
3      float currAccel, currGyro, currComplementary;
4      float sampleTime;
5
6      if (!IMU.gyroscopeAvailable()) return;
7      IMU.readGyroscope(gx, gy, gz);
8  }
```



```

9   if (!IMU.accelerationAvailable()) return;
10  IMU.readAcceleration(ax, ay, az);
11
12  currAccel = atan2(az, ay) * (180 / PI);
13  currAccel = (currAccel - 90) + ACCELEROMETER_OFFSET;
14
15  sampleTime = 1.0 / IMU.gyroscopeSampleRate();
16
17  currGyro = prevGyro + gx * sampleTime;
18
19  prevAngle = prevComplementary;
20  currComplementary = k * (prevComplementary + gx * sampleTime) + (1 - k) * currAccel;
21
22  /* Update Time Variables */
23  t_n = millis(); // Current Time in Milliseconds
24  dt = (t_n - t_n1) / 1000.0; // Time Difference in Seconds
25  t_n1 = t_n; // Assign Current Time to Previous Time
26
27  /* Update Angles */
28  Angles.Accelerometer = currAccel;
29  Angles.Gyroscope = currGyro;
30  Angles.Complementary = currComplementary;
31
32  /* Assign Previous Angles */
33  prevGyro = currGyro;
34  prevComplementary = currComplementary;
35 }

```

Listing 1: Arduino IMU Complementary Filter Firmware Implementation

```

1  void DistanceSensor_InputCaptureInterrupt(distancesensor* sensor)
2  {
3      if (HAL_GPIO_ReadPin(sensor->icGPIOPort, sensor->icGPIOPin)) {
4          sensor->IC_Value1 = HAL_TIM_ReadCapturedValue(sensor->timer, TIM_CHANNEL_1); // First
5              rising edge
6          HAL_TIM_PWM_Stop(sensor->timer, TIM_CHANNEL_2);
7      }
8      else {
9          sensor->IC_Value2 = HAL_TIM_ReadCapturedValue(sensor->timer, TIM_CHANNEL_1); // Second
10             rising edge
11             if (sensor->IC_Value2 > sensor->IC_Value1) {
12                 sensor->timeDifference = sensor->IC_Value2 - sensor->IC_Value1;
13             }
14             else {
15                 sensor->timeDifference = (TIM_PERIOD + 1 - sensor->IC_Value1) + sensor->IC_Value2;
16                 // Handle overflow
17             }
18
19             HAL_TIM_PWM_Start(sensor->timer, TIM_CHANNEL_2);
20             __HAL_TIM_SetCounter(sensor->timer, 65535);
21
22             DistanceSensor_Handle(sensor);
23         }
24     }
25 }
26
27 // On main.c:
28
29 void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
30     if (htim->Instance == TIM21) {
31         DistanceSensor_InputCaptureInterrupt(&Front);
32     }
33     else if (htim->Instance == TIM22) {
34         DistanceSensor_InputCaptureInterrupt(&Back);
35     }
36 }

```

Listing 2: STM32 HC-SR04 Firmware Implementation

```

1  #include "colorsensor.h"

```

```

2
3 #define TCS3472_SLAVE_ADDRESS 0x29 << 1
4 #define TCS3472_EXPECTED_ID 0x44
5
6 #define TCS3472_ID_REG 0x12
7
8 #define TCS3472_WTIME_REG 0x03
9 #define TCS3472_ATIME_REG 0xF6
10 #define TCS3472_ENABLE_REG 0x00
11
12 #define TCS3472_CDATAL_REG 0x14
13 #define TCS3472_RDATAL_REG 0x16
14 #define TCS3472_GDATAL_REG 0x18
15 #define TCS3472_BDATAL_REG 0x1A
16
17 extern speaker Speaker;
18
19 void ColorSensor_Init(colorsensor* sensor, I2C_HandleTypeDef* i2c_handle) {
20     sensor->i2c = i2c_handle;
21     sensor->slave_address = TCS3472_SLAVE_ADDRESS;
22
23     memset(sensor->rgb_data, 0, sizeof(sensor->rgb_data)); // Clear RGB data
24     sensor->enabled = true;
25     sensor->count = 0;
26
27     // Define initialization sequence
28     uint8_t init_data[][2] = {
29         { TCS3472_WTIME_REG | 0x80, 0xFF }, // Wait time
30         { TCS3472_ATIME_REG | 0x80, 0xFF }, // Integration time
31         { TCS3472_ENABLE_REG | 0x80, 0x0B }, // Power ON & enable RGBC
32     };
33
34     // Loop through initialization commands and send them
35     for (size_t i = 0; i < sizeof(init_data) / sizeof(init_data[0]); i++) {
36         if (HAL_I2C_Master_Transmit(sensor->i2c, sensor->slave_address, init_data[i], 2,
37             HAL_MAX_DELAY) != HAL_OK)
38             while(1);
39     }
40
41     HAL_GPIO_WritePin(COLOR_SENSOR_STATUS_GPIO_Port, COLOR_SENSOR_STATUS_Pin, GPIO_PIN_SET);
42 }
43
44 void ColorSensor_EnableStatus(colorsensor* sensor, bool enable)
45 {
46     sensor->enabled = enable;
47 }
48
49
50 uint16_t ColorSensor_Read16(colorsensor* sensor, uint8_t reg) {
51     uint8_t buffer[2];
52     uint8_t command = reg | 0xA0; // Add auto-increment bit
53
54     // Send register address
55     if (HAL_I2C_Master_Transmit(sensor->i2c, sensor->slave_address, &command, 1, HAL_MAX_DELAY)
56         != HAL_OK) {
57         return 0; // Handle error
58     }
59
60     // Read 2 bytes from the sensor
61     if (HAL_I2C_Master_Receive(sensor->i2c, sensor->slave_address, buffer, 2, HAL_MAX_DELAY) !=
62         HAL_OK) {
63         return 0; // Handle error
64     }
65
66     // Combine two bytes into a 16-bit value (LSB first)
67     return (uint16_t)(buffer[0] | (buffer[1] << 8));
68 }
69
70 void ColorSensor_ReadAll(colorsensor* sensor) {
71     sensor->rgb_data[0] = ColorSensor_Read16(sensor, TCS3472_CDATAL_REG); // TCS3472_CDATAL

```

```

70     sensor->rgb_data[1]    = ColorSensor_Read16(sensor, TCS3472_RDATAL_REG); // TCS3472_RDATAL
71     sensor->rgb_data[2]    = ColorSensor_Read16(sensor, TCS3472_GDATAL_REG); // TCS3472_GDATAL
72     sensor->rgb_data[3]    = ColorSensor_Read16(sensor, TCS3472_BDATAL_REG); // TCS3472_BDATAL
73 }
74
75
76 color ColorSensor_CalculateColor(colorsensor* sensor)
77 {
78     uint16_t max_val = sensor->rgb_data[1];
79     color detected_color = RED;
80
81     if (sensor->rgb_data[2] > max_val) detected_color = GREEN;
82     if (sensor->rgb_data[3] > max_val) detected_color = BLUE;
83
84     return detected_color;
85 }
86
87 void ColorSensor_Handle(colorsensor* sensor)
88 {
89     ColorSensor_ReadAll(sensor);
90     color detected_color = ColorSensor_CalculateColor(sensor);
91
92     if (detected_color == RED)
93     {
94         HAL_GPIO_WritePin(COLOR_SENSOR_STATUS_GPIO_Port, COLOR_SENSOR_STATUS_Pin,
95                             GPIO_PIN_RESET);
96         if (!Speaker.hasFault)
97             Speaker_Start(&Speaker, COLOR_SENSOR_ID);
98     }
99     else
100     {
101         HAL_GPIO_WritePin(COLOR_SENSOR_STATUS_GPIO_Port, COLOR_SENSOR_STATUS_Pin, GPIO_PIN_SET);
102         if (Speaker.hasFault)
103             Speaker_Stop(&Speaker, COLOR_SENSOR_ID);
104     }
105 }
106

```

Listing 3: STM32 TCS34725 Firmware Implementation

```

1  #include "rfid.h"
2
3  #include "speaker.h"
4
5  #define SERIALNUM_1_IDLE 32
6
7  #define SERIALNUM_0_CORRECT_CARD 170
8  #define SERIALNUM_1_CORRECT_CARD 205
9  #define SERIALNUM_2_CORRECT_CARD 47
10 #define SERIALNUM_3_CORRECT_CARD 3
11 #define SERIALNUM_4_CORRECT_CARD 75
12
13 extern speaker Speaker;
14 extern UART_HandleTypeDef huart1;
15 extern char Data;
16
17 void RFID_Init(rfid* sensor) {
18     MFRC522_Init();
19     memset(sensor->prevSerialNum, 0, 5);
20     sensor->status = CARD_IDLE;
21
22     sensor->botEnabled = false;
23     sensor->initialSuccessfulCardTap = true;
24     sensor->initialFailedCardTap = true;
25
26     HAL_GPIO_WritePin(RFID_STATUS_GPIO_Port, RFID_STATUS_Pin, GPIO_PIN_SET);
27 }
28
29 rfid_card_status RFID_ValidateCard(rfid* sensor)
30 {

```

```

31     uint8_t serialNum[5];
32     MFRC522_Request(PICC_REQIDL, serialNum);
33     MFRC522_Anticoll(serialNum);
34
35     sensor->status = CARD_IDLE;
36
37     if ((serialNum[0] == 170 && serialNum[1] == 205 && serialNum[2] == 47 && serialNum[3] == 3
38         && serialNum[4] == 75) ||
39         (sensor->prevSerialNum[0] == 170 && sensor->prevSerialNum[1] == 205 &&
40         sensor->prevSerialNum[2] == 47 && sensor->prevSerialNum[3] == 3 &&
41         sensor->prevSerialNum[4] == 75))
42     {
43         sensor->status = CARD_SUCCESS;
44     }
45     else if (!(serialNum[0] | (!(serialNum[1] == 32 || serialNum[1] == 0)) | serialNum[2] |
46         serialNum[3] | serialNum[4]))
47     {
48         if (!(sensor->prevSerialNum[0] | sensor->prevSerialNum[1] | sensor->prevSerialNum[2] |
49             sensor->prevSerialNum[3] | sensor->prevSerialNum[4]))
50             sensor->status = CARD_IDLE;
51         else
52         {
53             sensor->status = CARD_FAIL;
54         }
55     }
56     else
57     {
58         sensor->status = CARD_FAIL;
59     }
60
61     for (uint8_t i = 0; i < 5; i++)
62     {
63         sensor->prevSerialNum[i] = serialNum[i];
64     }
65
66     return sensor->status;
67 }
68
69 void RFID_SecurityLogic(rfid* sensor)
70 {
71     rfid_card_status cardStatus = RFID_ValidateCard(sensor);
72
73     switch (cardStatus) {
74         case CARD_SUCCESS:
75             if (sensor->initialSuccessfulCardTap)
76             {
77                 Speaker_SetAutoReload(&Speaker, 488);
78                 Speaker_Beep(&Speaker, 150, 0, 1);
79                 sensor->initialSuccessfulCardTap = false;
80                 sensor->initialFailedCardTap = true;
81                 sensor->botEnabled = !sensor->botEnabled;
82                 HAL_GPIO_WritePin(RFID_STATUS_GPIO_Port, RFID_STATUS_Pin, (GPIO_PinState)
83                     !sensor->botEnabled);
84             }
85             break;
86
87         case CARD_FAIL:
88             if (sensor->initialFailedCardTap)
89             {
90                 HAL_Delay(200);
91                 if (RFID_ValidateCard(sensor) != CARD_FAIL)
92                     break;
93                 Speaker_SetAutoReload(&Speaker, 488 * 4);
94                 Speaker_Beep(&Speaker, 150, 50, 4);
95                 sensor->initialSuccessfulCardTap = true;
96                 sensor->initialFailedCardTap = false;

```

```

96         }
97         break;
98
99
100     case CARD_IDLE:
101         sensor->initialSuccessfulCardTap = true;
102         sensor->initialFailedCardTap = true;
103         break;
104
105     // Optional: Default case if no case matches
106     default:
107         // Code to execute if none of the above cases match
108         break;
109 }
110
111 }

```

Listing 4: STM32 RC522 Firmware Implementation

```

1  #include "speaker.h"
2
3
4  #define CLK_SPEED 32000000
5  #define DEFAULT_AUTORELOAD 488
6
7  void Speaker_Init(speaker* speaker, rfid* rfid_struct, TIM_HandleTypeDef* timer)
8  {
9      speaker->rfid_sensor = rfid_struct;
10     speaker->timer = timer;
11
12     speaker->hasFault = false;
13     speaker->beepLengthOn = 0;
14     speaker->beepLengthPeriod = 0;
15     speaker->wantedNumBeeps = 0;
16     speaker->currentNumBeeps = 0;
17     speaker->timerCounter = 0;
18
19     for (uint8_t i = 0; i < sizeof(speaker->featureFault) / sizeof(speaker->featureFault[0]);
20         i++)
21     {
22         speaker->featureFault[i] = false;
23     }
24
25 }
26
27 void Speaker_Start(speaker* speaker, uint8_t ID)
28 {
29
30     speaker->featureFault[ID] = true;
31     if ((speaker->featureFault[0] || speaker->featureFault[1] || speaker->featureFault[2]) &&
32         speaker->rfid_sensor->botEnabled)
33     {
34         speaker->hasFault = true;
35         Speaker_SetAutoReload(speaker, DEFAULT_AUTORELOAD);
36         HAL_TIM_PWM_Start(speaker->timer, TIM_CHANNEL_1);
37     }
38 }
39
40 void Speaker_Stop(speaker* speaker, uint8_t ID)
41 {
42     speaker->featureFault[ID] = false;
43     if (!(speaker->featureFault[0] || speaker->featureFault[1] || speaker->featureFault[2]))
44     {
45         speaker->hasFault = false;
46         HAL_TIM_PWM_Stop(speaker->timer, TIM_CHANNEL_1);
47     }
48 }
49
50

```

```

51
52 bool Speaker_Beep(speaker* speaker, uint16_t length_on_ms, uint16_t length_off_ms, uint8_t
    numBeeps)
53 {
54     if (speaker->hasFault)
55         return false;
56
57
58
59     speaker->timerCounter = 0;
60     speaker->currentNumBeeps = 0;
61
62
63
64     speaker->beepLengthOn = length_on_ms * 1000 / __HAL_TIM_GET_AUTORELOAD(speaker->timer);
65     speaker->beepLengthPeriod = speaker->beepLengthOn + length_off_ms * 1000 / __
        HAL_TIM_GET_AUTORELOAD(speaker->timer);
66     speaker->wantedNumBeeps = numBeeps;
67
68     //speaker->beepLength = length_ms * CLK_SPEED / (speaker->timer->Instance->PSC);
69
70     HAL_TIM_Base_Start_IT(speaker->timer);
71     HAL_TIM_PWM_Start(speaker->timer, TIM_CHANNEL_1);
72     return true;
73 }
74
75 void Speaker_BeepInterrupt(speaker* speaker)
76 {
77     if (speaker->currentNumBeeps < speaker->wantedNumBeeps)
78     {
79
80         if (speaker->timerCounter == speaker->beepLengthOn)
81         {
82             HAL_TIM_PWM_Stop(speaker->timer, TIM_CHANNEL_1);
83             __HAL_TIM_ENABLE(speaker->timer);
84         }
85         else if (speaker->timerCounter >= speaker->beepLengthPeriod)
86         {
87             speaker->currentNumBeeps++;
88             speaker->timerCounter = 0;
89
90             if (speaker->currentNumBeeps < speaker->wantedNumBeeps)
91             {
92                 HAL_TIM_PWM_Start(speaker->timer, TIM_CHANNEL_1);
93
94             }
95
96         }
97         speaker->timerCounter++;
98     }
99     else
100     {
101         HAL_TIM_PWM_Stop(speaker->timer, TIM_CHANNEL_1);
102         HAL_TIM_Base_Stop_IT(speaker->timer);
103     }
104 }
105
106 void Speaker_SetAutoReload(speaker* speaker, uint16_t value)
107 {
108     __HAL_TIM_SET_AUTORELOAD(speaker->timer, value);
109     __HAL_TIM_SET_COMPARE(speaker->timer, TIM_CHANNEL_1, value / 2);
110 }

```

Listing 5: STM32 Speaker Firmware Implementation

```

1     import asyncio
2     import threading
3
4     from bleak import BleakScanner, BleakClient, BleakError
5     from flask import Flask, request, jsonify, render_template
6

```

```

7 class BLEManager:
8     # Define UUIDs for BLE Service and Characteristic
9     SERVICE_UUID = "00000000-5EC4-4083-81CD-A10B8D5CF6EC"
10    CHARACTERISTIC_UUID = "00000001-5EC4-4083-81CD-A10B8D5CF6EC"
11    CODE = "0095" # Authentication Code
12
13    def __init__(self):
14        self.devices = []
15        self.client = None
16        self.is_connected = False
17
18        self.loop = asyncio.new_event_loop()
19        self.start_loop()
20
21    def start_loop(self):
22        threading.Thread(target = self._run_loop, daemon = True).start()
23
24    def _run_loop(self):
25        asyncio.set_event_loop(self.loop)
26        self.loop.run_forever()
27
28    def run_async(self, coro):
29        return asyncio.run_coroutine_threadsafe(coro, self.loop)
30
31    async def scan_devices(self):
32        self.devices = []
33        scanned = await BleakScanner.discover()
34
35        self.devices = [
36            {"name" : device.name, "address" : device.address} for device in scanned if
37                device.name and ("WALL-E" in device.name)
38        ] # Filter BLE Devices
39        return self.devices
40
41    async def connect_device(self, address):
42        """Connect to BLE Device."""
43        if self.client and self.client.is_connected:
44            self.is_connected = True
45            return True
46
47        self.client = BleakClient(address)
48        try:
49            await self.client.connect()
50            await self.client.write_gatt_char(BLEManager.CHARACTERISTIC_UUID,
51                BLEManager.CODE.encode("utf-8"))
52            self.is_connected = self.client.is_connected
53            return self.is_connected
54        except BleakError:
55            self.is_connected = False
56            return False
57
58    async def disconnect_device(self):
59        """Disconnect from BLE Device."""
60        if self.client and self.client.is_connected:
61            await self.client.disconnect()
62            self.is_connected = False
63
64    async def send_cmd(self, cmd):
65        """Schedule Send Command Coroutine Using Button's Custom Command."""
66        if not (self.client and self.client.is_connected):
67            return False, "Not Connected to BLE"
68        try:
69            await self.client.write_gatt_char(BLEManager.CHARACTERISTIC_UUID,
70                cmd.encode("utf-8"))
71            return True, f"Sent Command: {cmd}"
72        except BleakError as e:
73            return False, f"Error Sending Command: {e}"
74
75 class RobotDriverApp:
76     def __init__(self):
77         self.app = Flask("Robot Driver App")

```

```

75         self.ble_manager = BLEManager()
76         self._register_routes()
77
78     def _register_routes(self):
79         @self.app.route("/")
80         def index():
81             return render_template("index.html")
82
83         @self.app.route("/scan", methods = ["GET"])
84         def scan():
85             future = self.ble_manager.run_async(self.ble_manager.scan_devices())
86             devices = future.result(timeout = 30) # Wait for 30 Seconds
87             return jsonify(devices)
88
89         @self.app.route("/connect", methods = ["POST"])
90         def connect():
91             data = request.get_json()
92             if not data or "deviceAddress" not in data:
93                 return jsonify({"error": "No Address Provided"}), 400
94             future =
95                 self.ble_manager.run_async(self.ble_manager.connect_device(data["deviceAddress"]))
96             success = future.result(timeout = 15) # Wait for 15 Seconds
97             return (jsonify({"status": "Connected"}) if success
98                     else (jsonify({"status": "Failed"}), 400))
99
100        @self.app.route("/disconnect", methods = ["GET"])
101        def disconnect():
102            future = self.ble_manager.run_async(self.ble_manager.disconnect_device())
103            future.result(timeout = 10) # Wait for 5 Seconds
104            return jsonify({"status": "Disconnected"})
105
106        @self.app.route("/move", methods = ["POST"])
107        def move():
108            data = request.get_json()
109            if not data or "command" not in data:
110                return jsonify({"error": "No Command Provided"}), 400 # Bad Request
111
112            future = self.ble_manager.run_async(
113                self.ble_manager.send_cmd(data["command"])
114            )
115            success, message = future.result(timeout = 10) # Wait for 5 Seconds
116
117            if success:
118                return jsonify({"status": "OK", "msg": message})
119            else:
120                return jsonify({"status": "Error", "msg": message}), 400 # Bad Request
121
122        def run(self):
123            self.app.run(host = "0.0.0.0", port = 5000, debug = False)
124
125    if __name__ == "__main__":
126        RobotDriverApp().run()

```

Listing 6: Robot Driver App

```

1  from picamera2 import Picamera2
2  from picamera2.encoders import H264Encoder
3  from libcamera import Transform
4
5  class CameraDisplay(tk.Frame):
6      SAMPLE_RATE = 1.0 / 50.0 # Update Delay ~20 FPS (1000ms/20 = 50ms)
7      def __init__(self, parent, *args, **kwargs):
8          super().__init__(parent, *args, **kwargs)
9
10         self.filename = ""
11         self.camera = Picamera2() # Init Camera
12
13         self.snapshot_dir: str = os.path.join(os.path.dirname(os.path.abspath(__file__)),
14             "Snapshots")
15         self.video_dir: str = os.path.join(os.path.dirname(os.path.abspath(__file__)), "Videos")

```



```

16         os.makedirs(self.snapshot_dir, exist_ok = True)
17         os.makedirs(self.video_dir, exist_ok = True)
18
19         # Force RGB888 Output
20         self.config = self.camera.create_video_configuration(
21             main = {
22                 "size": (640, 480), # Default Size
23                 # "size": (1280, 720), # HD
24                 # "size": (1920, 1080), # Full HD
25                 "format": "RGB888"
26             },
27             transform = Transform(hflip = False, vflip = False) # For Tkinter Display
28         )
29         self.image_label = tk.Label(self)
30         self.image_label.pack()
31
32     def start_recording(self):
33         self.camera.configure(self.config)
34         self.camera.start()
35
36         self.filename = f"Recording_{dt.datetime.now().strftime('%Y%m%d_%H%M%S')}"
37         input_file = os.path.join(self.video_dir, f"{self.filename}.h264")
38
39         self.camera.start_recording(H264Encoder(bitrate = 10000000), input_file)
40
41     def stop_recording(self):
42         if self.filename == "":
43             return
44
45         self.camera.stop_recording()
46         self.camera.stop()
47         self.filename = ""
48
49     def take_snapshot(self):
50         if self.filename == "":
51             self.camera.configure(self.config)
52             self.camera.start()
53
54         # Capture Current Frame
55         frame = self.camera.capture_array()
56         if frame is None or frame.size == 0:
57             print("Failed to Capture Snapshot.")
58             return
59
60         # Swap Color Channels (BGR -> RGB)
61         frame = frame[..., ::-1]
62
63         snapshot_name = f"Snapshot_{dt.datetime.now().strftime('%Y%m%d_%H%M%S')}.jpg"
64         snapshot_path = os.path.join(self.snapshot_dir, snapshot_name)
65
66         # Convert to PIL Image
67         image = PILImage.fromarray(frame)
68         image.save(snapshot_path) # Save Image
69         print(f"Snapshot Saved to {snapshot_path}")
70
71         if self.filename == "":
72             self.camera.stop()
73
74     def update(self):
75         if self.filename == "":
76             return
77
78         frame = self.camera.capture_array()
79         if (frame is not None) and (frame.size != 0):
80             frame = frame[..., ::-1] # BGR -> RGB
81
82             # Convert the PIL Image to an ImageTk.PhotoImage
83             self.image_label.current = ImageTk.PhotoImage(
84                 image = PILImage.fromarray(frame) # Convert Numpy Array to PIL.Image
85             )
86             self.image_label.configure(image = self.image_label.current)

```

```
87
88     def stop(self):
89         self.camera.close()
```

Listing 7: PiCamera Display