
# Complex Engineering Problem (CEP)

**Submitted to:** Engr Abu Bakar Talha Jalil


| Name | Registration No. | Email ID |
|---|---|---|
| M. Naeem | FA21-BCE-100 | fa21-bce-100@cuilahore.edu.pk |
| M. Subhan Saleem | FA21-BCE-056 | Fa21-bce-056@cuilahore.edu.pk |
| Rizwan Ali | FA21-BCE-078 | fa21-bce-078@cuilahore.edu.pk |
| M. Faseeh Khan | FA21-BCE-011 | Fa21-bce-011@cuilahore.edu.pk |

# Abstract

This project presents the design, development, and implementation of an autonomous Line Following Robot with 90° Turn Detection using an ESP32 microcontroller, an IR sensor strip, and a PID control algorithm. The robot is capable of accurately tracking a black line on a white surface while maintaining stability at varying speeds. A robust turn-detection mechanism enables reliable identification and execution of sharp 90-degree turns, ensuring smooth navigation on complex tracks. Additionally, the system incorporates an effective line-loss recovery strategy to enhance operational reliability.

To improve real-time performance, FreeRTOS is employed to divide sensor acquisition and motor control into concurrent tasks, resulting in faster response time and improved control accuracy. This project demonstrates the practical integration of embedded systems**,** control theory (PID)**,** real-time operating systems, and sensor-based robotics, providing hands-on experience with modern autonomous robotic system design.

**Table of Contents**

## Tables OF Figure

# 1. Introduction

The objective of this CEP (Course Engineering Project) is to design and build an autonomous **line following robot** capable of handling straight paths, curves, and sharp 90-degree turns. Line following robots are widely used in industrial automation, warehouses, and autonomous guided vehicles (AGVs). This project focuses on improving accuracy and reliability by using **PID control** and **FreeRTOS-based multitasking** on an ESP32.

The robot detects a black line on a white surface using an IR sensor strip and controls the motors accordingly. Special logic is implemented to detect 90° turns, allowing the robot to navigate complex tracks.

# 2. System Overview

The system consists of three main subsystems:

- **Sensing System:** The 5-IR sensor strip detects the position of the black line.
- **Control System:** The ESP32 microcontroller processes sensor data using PID control, executes 90° turn detection logic, and manages FreeRTOS tasks for real-time performance.
- **Actuation System:** The motor driver controls the left and right motors based on the signals from the ESP32 to ensure accurate line following.

The robot continuously reads sensor data, calculates the line position error, and adjusts motor speeds to stay aligned with the line. Special logic is implemented to detect sharp 90° turns and recover from line-loss situations.

The block diagram illustrates the flow of information and power in the system. The battery and buck converter provide power to all components. The IR sensors send readings to the ESP32, where line position calculation, PID control, and 90° turn detection are processed. FreeRTOS tasks manage sensor reading and motor control concurrently. Control signals from the ESP32

drive the motors through the motor driver. The OLED display receives status and data for debugging and monitoring. Feedback from the motors helps maintain accurate line tracking.

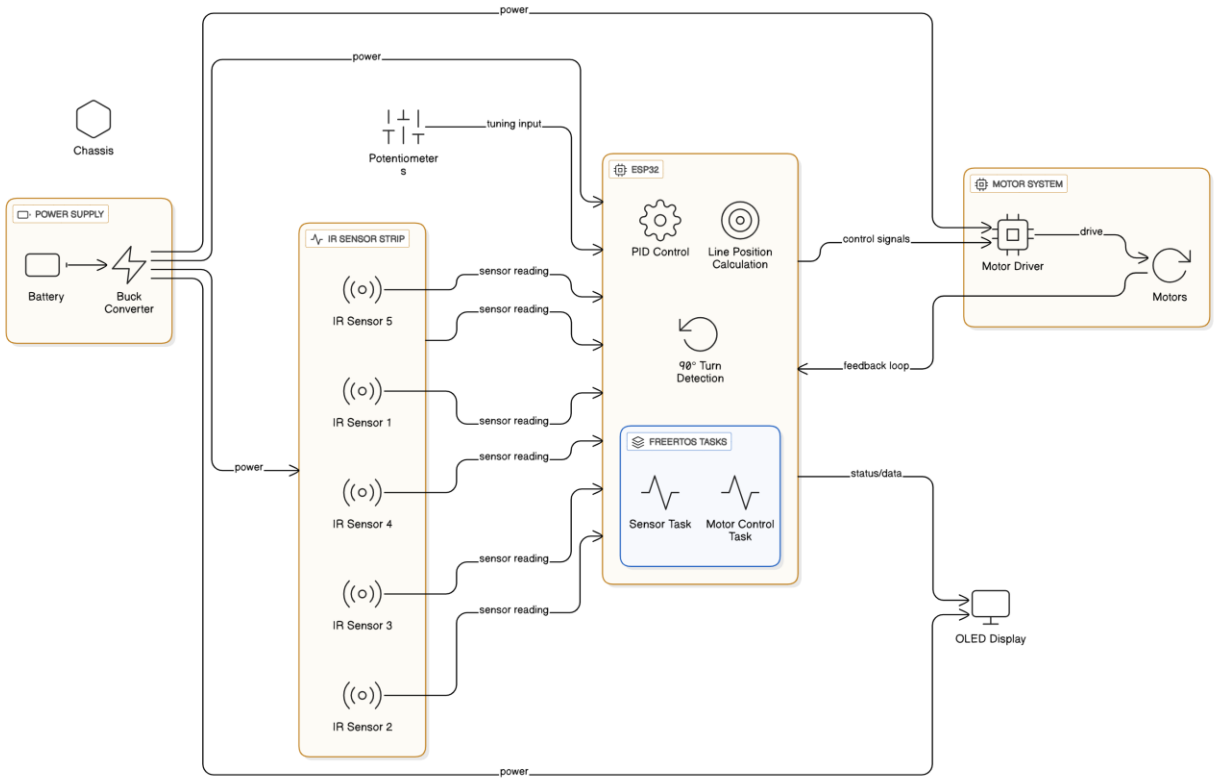


*Figure 1 Block Diagram of the Autonomous Line Following Robot*

# 3. Hardware Components Description

- **Chassis 2000**

  Provides mechanical structure to mount motors, sensors, ESP32, and battery.
- **Battery (3 × 10,000 mAh Cells)**

  Supplies sufficient power for motors and control electronics.
- **Buck Converter**

  Steps down battery voltage to safe levels for ESP32 and sensors.

- **ESP32 Microcontroller**

  Acts as the brain of the robot. Handles sensor processing, PID control, motor driving, and FreeRTOS task management.
- **IR Sensor Strip (5 Sensors)**

  Detects the black line. Sensor logic used:

  `1` = Black line

  0 = White surface
- **Motor Driver & Motors**

  Controls direction and speed of left and right motors using PWM.
- **Potentiometers (3)**

  Used for calibration and tuning (e.g., sensor sensitivity or speed adjustment).
- **OLED Display**

  Displays debugging information such as sensor status or robot state.
- **Breadboard and Wires**

  Used for prototyping and interconnections.

# 4. Software Design and Algorithm

## 4.1. FreeRTOS Task Structure

Two FreeRTOS tasks are used:

- **Sensor Task (Core 0)**
  - Continuously reads IR sensors and computes the line position.
- **Motor Control Task (Core 1)**
  - Handles PID control, motor driving, line loss recovery, and 90° turn execution.
  - A mutex is used to safely share sensor data between tasks.

## 4.2. Line Position Calculation

Each sensor is assigned a weight:

| Sensor | Position Weight |
|---|---|
| S1 (Leftmost) | -2 |
| S2 | -1 |
| S3 (Center) | 0 |
| S4 | +1 |
| S5 (Rightmost) | +2 |

The weighted average of active sensors determines the line position error.

## 4.3. PID Control

PID control is used to smoothly follow the line:

- **Proportional (P):** Corrects current error
- **Integral (I):** Eliminates steady-state error
- **Derivative (D):** Reduces oscillations

Motor speeds are adjusted as:

- Left Speed = Base Speed − Correction
- Right Speed = Base Speed + Correction

## 4.4. 90° Turn Detection Logic

A 90° turn is detected when **three or more consecutive sensors** on one side detect black.

- **Left Turn:** S3, S4, S5 = 1
- **Right Turn:** S1, S2, S3 = 1

Once detected:

- Robot moves forward slightly.
- Executes in-place rotation.
- Exits turn mode when center sensor detects the line again.

## 4.5. Line Loss Recovery

If all sensors read white:

- Robot rotates in the direction where the line was last seen.
- PID integral is reset.

# 5. Integration and Working Procedure

All hardware components were assembled on the chassis. Sensors were placed at the front, close to the ground. The ESP32 was programmed using Arduino IDE. Motor driver pins were connected to ESP32 PWM-capable pins.

After uploading the code:

- Sensors detect the line
- ESP32 calculates error
- PID adjusts motor speed
- Special logic handles sharp turns and recovery

# 6. Testing and Results

The developed line following robot was tested on a white surface with a black electrical tape track. The test track consisted of straight paths, smooth curves, and sharp 90° turns. The

objective of testing was to evaluate sensor performance, PID controller behavior, motor response, and the effectiveness of the 90° turn detection algorithm.

To analyze system performance in detail, **MATLAB was used to simulate and visualize sensor readings, line position error, motor speeds, and PID correction output**. The graphical results obtained from MATLAB are discussed below.

## 6.1. Graphical Analysis of System Performance

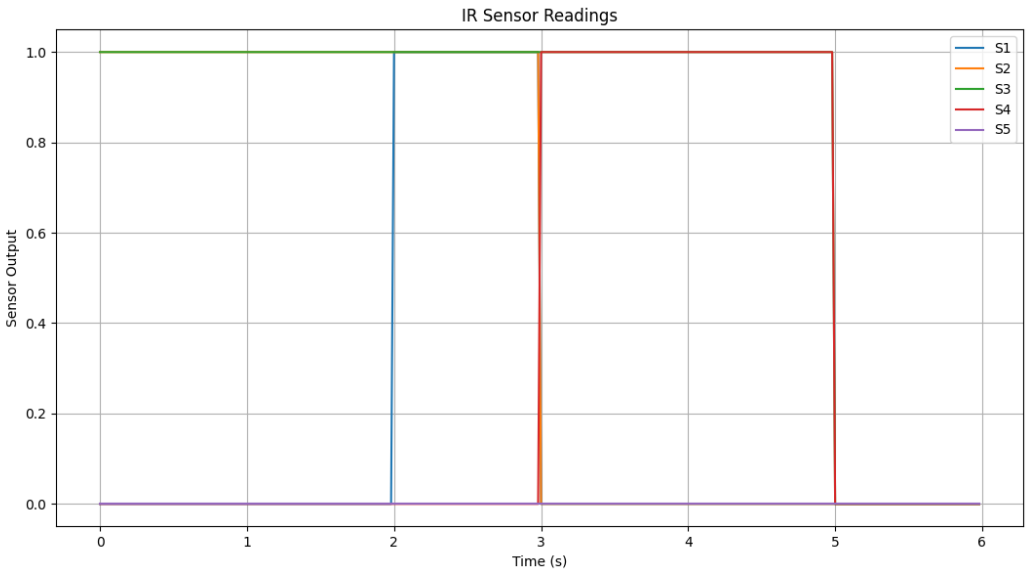


*Figure 2 IR Sensor Output Patterns During Line Following and 90° Turns*

This graph shows the output of the five IR sensors (S1 to S5) with respect to time. A value of **1 indicates detection of the black line**, while **0 represents the white surface**. During straight-line motion, the center sensors remain active, whereas during 90° turns, three or more adjacent sensors on one side become active. This behavior confirms accurate sensor placement and reliable detection of line patterns required for turn identification.

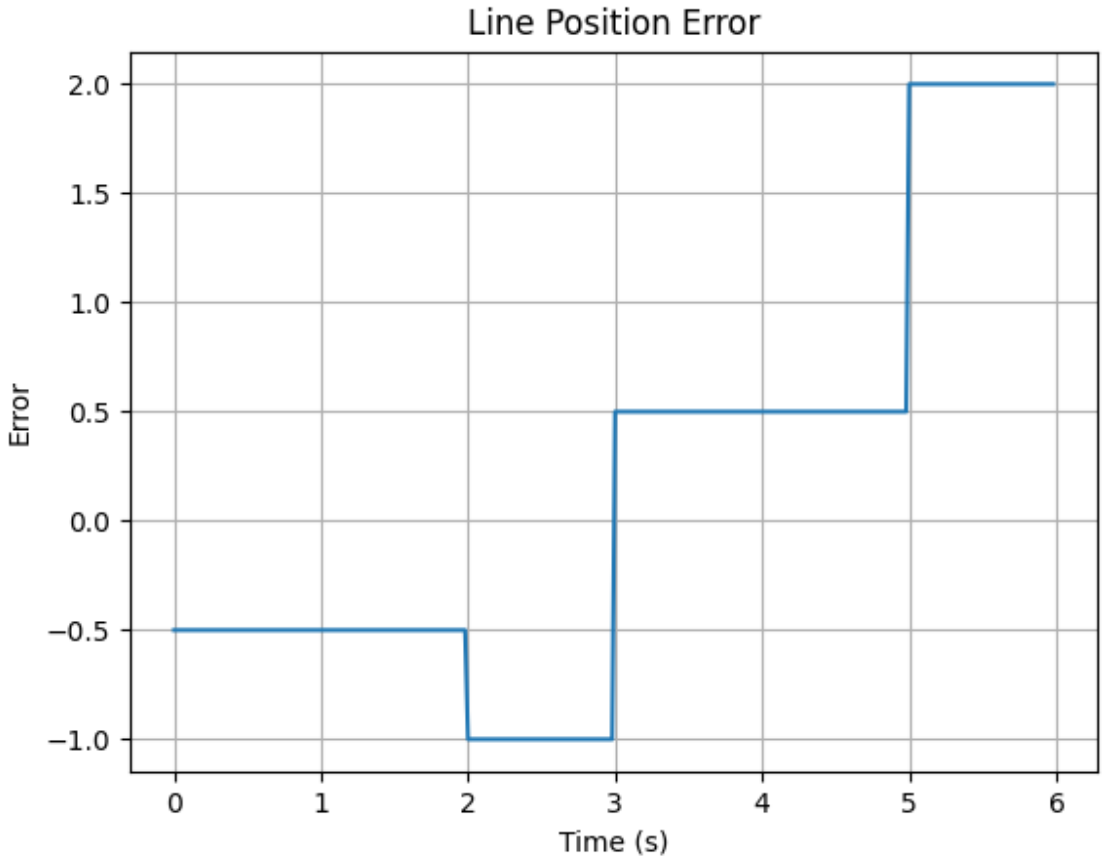


*Figure 3 Line Position Error Variation with Time*

This graph represents the variation of line position error calculated using the weighted average method. When the robot is aligned with the line, the error remains close to zero. During curves and 90° turns, a sudden increase in error is observed, followed by a rapid correction. This demonstrates the effectiveness of the line position calculation and the responsiveness of the control system.

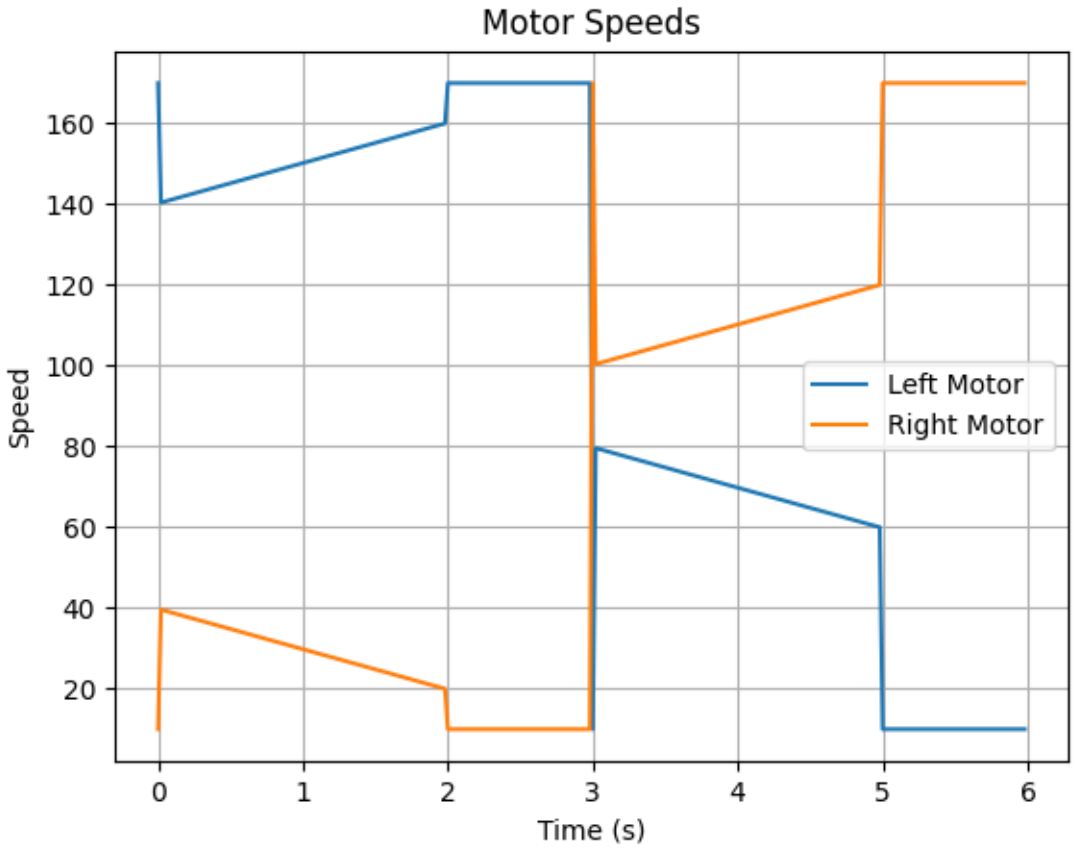


*Figure 4 Left and Right Motor Speed Response Under PID Control*

This graph illustrates the speed variation of the left and right motors as controlled by the PID algorithm. When the robot deviates from the line, the motor speeds adjust differentially to correct the path. During sharp turns, one motor slows down while the other accelerates, enabling smooth turning behavior. The symmetry in speed recovery indicates stable and balanced motor control.

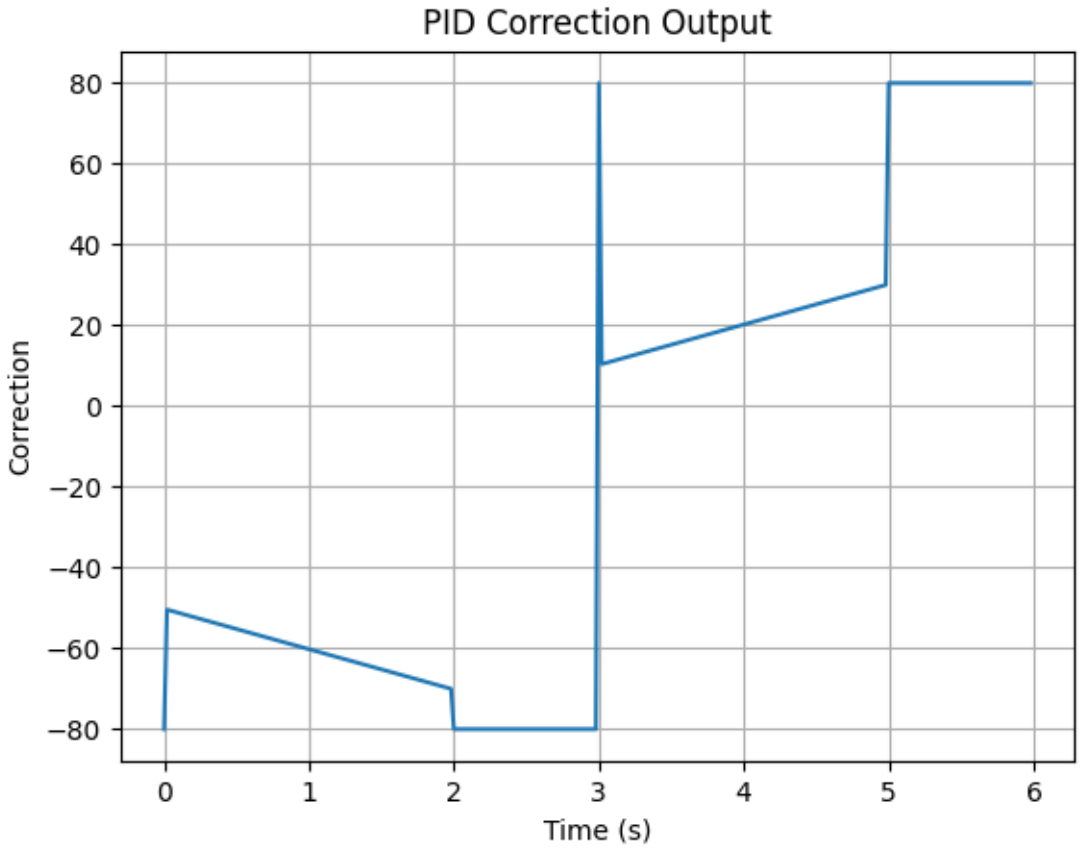


*Figure 5 PID Control Signal Generated Using MATLAB*

This graph shows the overall PID correction signal applied to the motors. Higher correction values are observed during sharp turns and line loss scenarios, while lower values occur during straight-line motion. The bounded correction output confirms proper tuning of PID parameters, preventing excessive oscillations and ensuring stable robot movement.

## 6.2. Discussion of Results

The MATLAB-based graphical analysis verifies that the implemented PID controller successfully minimizes line tracking error while maintaining smooth motor operation. The IR sensor graphs clearly indicate correct detection of straight paths and 90° turns. The motor speed and PID correction plots demonstrate stable performance, quick response to disturbances, and effective recovery from line loss conditions.

Overall, the testing results confirm that the system meets the design objectives of accurate line following, reliable 90° turn detection, and robust real-time control using FreeRTOS.

# 7. Conclusion

This Course Engineering Project successfully achieved the design and implementation of an autonomous line following robot capable of accurately tracking a black line on a white surface while handling sharp 90° turns and line loss conditions. The integration of an IR sensor strip with an ESP32 microcontroller enabled reliable sensing and real-time decision making.

The use of a PID control algorithm ensured smooth and stable line tracking by continuously minimizing positional error and adjusting motor speeds accordingly. Furthermore, the implementation of a robust 90° turn detection mechanism allowed the robot to navigate complex tracks without manual intervention. The incorporation of FreeRTOS significantly improved system responsiveness by enabling concurrent execution of sensor acquisition and motor control tasks.

Experimental testing and MATLAB-based graphical analysis validated the effectiveness of the proposed system. The results demonstrated accurate line detection, stable motor control, quick recovery from disturbances, and reliable execution of sharp turns. Overall, this project provided valuable practical experience in embedded systems, real-time operating systems, control algorithms, and autonomous robotics, fulfilling all stated project objectives.

# 8. Limitations and Future Enhancements

## 8.1. Limitations

Despite successful implementation, the system has certain limitations:

- The performance of IR sensors may be affected by ambient lighting conditions and surface reflectivity.
- Fixed PID parameters may not provide optimal performance for all track layouts and speeds.

- The robot's turning accuracy depends on precise sensor alignment and consistent track width.
- Battery voltage fluctuations can influence motor speed and overall system stability.
- The system lacks environmental awareness beyond line detection.

## 8.2. Future Enhancements

Several improvements can be made to enhance the system's functionality and robustness:

- Implementation of adaptive or self-tuning PID control to automatically adjust parameters.
- Integration of wireless communication (Wi-Fi or Bluetooth) for real-time monitoring and control.
- Addition of a camera-based vision system for advanced path recognition.
- Data logging and performance analysis using onboard memory or cloud storage.
- Incorporation of obstacle detection sensors to enable autonomous navigation in dynamic environments.

These enhancements would increase the intelligence, reliability, and application scope of the line following robot in real-world industrial and autonomous systems.

# 9. Appendices

## 9.1. Appendix A: Header Files, Pin Configuration, and Global Parameters

```
// ================================================
// ESP32 Line Follower with 90° Turn Detection
// 5 IR Sensors + FreeRTOS + PID Control
// ================================================
```

```
#include <Arduino.h>

// -------------------- SENSOR PINS --------------------
#define S1_PIN 25 // Leftmost
#define S2_PIN 33 // Left-Center
#define S3_PIN 32 // Center
#define S4_PIN 35 // Right-Center
#define S5_PIN 34 // Rightmost

// -------------------- MOTOR PINS --------------------
#define IN1_PIN 12 // Left Motor, Pin 1
#define IN2_PIN 13 // Left Motor, Pin 2
#define IN3_PIN 27 // Right Motor, Pin 1
#define IN4_PIN 26 // Right Motor, Pin 2

// ---------- PID PARAMETERS ----------
float Kp = 100.0;
float Ki = 20.0;
float Kd = 12.0;
int baseSpeed = 90;
int maxCorrection = 80;

// ---------- 90° TURN PARAMETERS ----------
```

```
int turn90Speed = 110;

int turn90MinTime = 150;

int moveForwardBeforeTurn = 50;


// ---------- GLOBAL VARIABLES ----------

volatile int s1, s2, s3, s4, s5;

volatile float linePos = 0.0;

volatile int lastDirection = 1;

volatile bool turn90Active = false;

volatile int turn90Direction = 0;

volatile unsigned long turn90StartTime = 0;


float prevError = 0;

float integral = 0;

unsigned long prevTime = 0;


SemaphoreHandle_t mutex;
```

## 9.2. Appendix B: Motor Control Function

This appendix contains the function responsible for controlling the direction and speed of both motors using PWM signals.

```
void driveMotors(int leftSpeed, int rightSpeed) {

  leftSpeed = constrain(leftSpeed, -255, 255);
```

```
rightSpeed = constrain(rightSpeed, -255, 255);

// Left Motor Control
if (leftSpeed > 0) {
  analogWrite(IN1_PIN, leftSpeed);
  analogWrite(IN2_PIN, 0);
} else if (leftSpeed < 0) {
  analogWrite(IN1_PIN, 0);
  analogWrite(IN2_PIN, abs(leftSpeed));
} else {
  analogWrite(IN1_PIN, 0);
  analogWrite(IN2_PIN, 0);
}

// Right Motor Control
if (rightSpeed > 0) {
  analogWrite(IN3_PIN, rightSpeed);
  analogWrite(IN4_PIN, 0);
} else if (rightSpeed < 0) {
  analogWrite(IN3_PIN, 0);
  analogWrite(IN4_PIN, abs(rightSpeed));
} else {
  analogWrite(IN3_PIN, 0);
```

```
    analogWrite(IN4_PIN, 0);
  }
}
```

## 9.3. Appendix C: System Initialization and FreeRTOS Setup

This appendix includes the system initialization routine and creation of FreeRTOS tasks.

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

  pinMode(S1_PIN, INPUT);
  pinMode(S2_PIN, INPUT);
  pinMode(S3_PIN, INPUT);
  pinMode(S4_PIN, INPUT);
  pinMode(S5_PIN, INPUT);

  pinMode(IN1_PIN, OUTPUT);
  pinMode(IN2_PIN, OUTPUT);
  pinMode(IN3_PIN, OUTPUT);
  pinMode(IN4_PIN, OUTPUT);

  driveMotors(0, 0);
```

```
  mutex = xSemaphoreCreateMutex();

  xTaskCreatePinnedToCore(readSensors, "Sensors", 4096, NULL, 2, NULL, 0);
  xTaskCreatePinnedToCore(controlMotors, "Motors", 4096, NULL, 1, NULL, 1);

  prevTime = millis();
}
```

## 9.4. Appendix D: Sensor Reading Task (FreeRTOS Task 1)

This appendix contains the task responsible for reading IR sensors and calculating line position.

```
void readSensors(void *param) {
  while (true) {
    int raw1 = 1 - digitalRead(S1_PIN);
    int raw2 = 1 - digitalRead(S2_PIN);
    int raw3 = 1 - digitalRead(S3_PIN);
    int raw4 = 1 - digitalRead(S4_PIN);
    int raw5 = 1 - digitalRead(S5_PIN);

    if (xSemaphoreTake(mutex, portMAX_DELAY)) {
      s1 = raw1; s2 = raw2; s3 = raw3; s4 = raw4; s5 = raw5;

      int blackCount = s1 + s2 + s3 + s4 + s5;
```

```
      if (blackCount == 0) {
        linePos = (lastDirection > 0) ? 2.0 : -2.0;
      } else {
        int weightedSum = (-2*s1) + (-1*s2) + (0*s3) + (1*s4) + (2*s5);
        linePos = (float)weightedSum / blackCount;

        if (linePos > 0.2) lastDirection = 1;
        if (linePos < -0.2) lastDirection = -1;
      }
      xSemaphoreGive(mutex);
    }
    vTaskDelay(5 / portTICK_PERIOD_MS);
  }
}
```

## 9.5. Appendix E: Motor Control and 90° Turn Detection Task (FreeRTOS Task 2)

This appendix implements PID control, sharp turn detection, and line loss recovery.

```
void controlMotors(void *param) {
  while (true) {
    int sensors[5];
```

```
float error;
int direction;

if (xSemaphoreTake(mutex, portMAX_DELAY)) {
  sensors[0]=s1; sensors[1]=s2; sensors[2]=s3;
  sensors[3]=s4; sensors[4]=s5;
  error = linePos;
  direction = lastDirection;
  xSemaphoreGive(mutex);
}

unsigned long now = millis();

bool rightTurn90 = (sensors[0] && sensors[1] && sensors[2]);
bool leftTurn90  = (sensors[2] && sensors[3] && sensors[4]);

if (!turn90Active && (leftTurn90 || rightTurn90)) {
  turn90Active = true;
  turn90Direction = leftTurn90 ? -1 : 1;
  turn90StartTime = now;
  driveMotors(baseSpeed, baseSpeed);
  delay(moveForwardBeforeTurn);
}
```

```
if (turn90Active) {
  driveMotors(turn90Direction * turn90Speed,
              -turn90Direction * turn90Speed);

  if ((now - turn90StartTime) > turn90MinTime) {
    int blackCount = sensors[0]+sensors[1]+sensors[2]+sensors[3]+sensors[4];
    if (blackCount <= 2 && sensors[2]) {
      turn90Active = false;
      integral = 0;
      prevError = 0;
    }
  }
  vTaskDelay(10 / portTICK_PERIOD_MS);
  continue;
}

if (sensors[0]+sensors[1]+sensors[2]+sensors[3]+sensors[4] == 0) {
  driveMotors(-direction*120, direction*120);
  integral = 0;
  vTaskDelay(10 / portTICK_PERIOD_MS);
  continue;
}
```

```
    float dt = (now - prevTime) / 1000.0;
    if (dt <= 0) dt = 0.01;

    float P = Kp * error;
    integral = constrain(integral + error*dt, -5, 5);
    float I = Ki * integral;
    float D = Kd * (error - prevError) / dt;

    float correction = constrain(P + I + D, -maxCorrection, maxCorrection);

    driveMotors(baseSpeed - correction, baseSpeed + correction);

    prevError = error;
    prevTime = now;

    vTaskDelay(20 / portTICK_PERIOD_MS);
  }
}
```