RobotZero A Comprehensive Environment for Line-Following Robot Development

This project provides a foundational software platform for developing line-following robots.

Frank Coelho de Alcantara

2024-10-11

Contents

1 Introduction

A line following robot is an autonomous system designed to follow a path marked by a line on the ground. These robots are often considered an entry point into robotics and automation, as they incorporate fundamental concepts of sensor reading, motor control, and real-time decision making. However, while the basic concept might seem simple, developing a high-performance line follower requires sophisticated control systems and precise calibration.

Named **RobotZero**, this project embodies the concept of starting from zero - both for those beginning their journey in robotics and in homage to DeepMind's AlphaGo Zero. Like its namesake, which learned chess and Go from scratch, achieving mastery through self-play and analysis, RoboTZero is designed to be a foundation for learning through systematic data collection and analysis. Despite its diminutive size and seemingly simple purpose, the robot incorporates sophisticated logging and analysis capabilities that transform it from a basic line follower into a platform for understanding robotics fundamentals, control systems, and performance optimization. It represents "zero" not as nothing, but as the essential starting point for building knowledge and expertise in robotics.

1.1 The Platform: Arduino Nano

For this project, we chose the Arduino Nano as our main controller specifically for its constrained environment. When we can achieve efficiency and speed within such limitations, we learn how to optimize ideas, algorithms, and programs - skills that are valuable across all computing platforms. The Arduino Nano, based on the ATmega328P microcontroller, offers a small form factor (45 x 18 mm), ideal for compact robot designs. It includes 32KB of Flash memory, suitable for our advanced logging system, 2KB of SRAM for runtime operations, and runs at 16MHz, providing adequate processing power. The board features multiple analog inputs for our sensor array, PWM outputs for precise motor control, and maintains low power consumption while being cost-effective for both prototyping and final implementation.

1.2 Project Overview

In most projects, adjusting a line following robot for speed and efficiency becomes a tedious and time-consuming trial-and-error routine. Our project addresses this challenge by incorporating an advanced logging and analysis system that transforms the tuning process into a data-driven approach. The robot includes high-precision line detection using a 6-sensor array, PID-based motion control for smooth operation, and dual operating modes for analysis and high-speed performance. The comprehensive data logging system enables real-time performance monitoring, with flash-based storage for post-run analysis and a USB interface for data retrieval and analysis, providing the tools necessary for systematic testing and configuration.

1.3 Why C++?

The choice of C++ as our programming language was deliberate and based on several key factors. C++ allows us to organize our code into logical classes and modules, making the system more maintainable and easier to understand. This is particularly important for complex systems like our logging mechanism. The language provides low-level hardware access while supporting high-level abstractions, crucial for real-time operations where microseconds matter, such as sensor reading and motor control.

With limited resources on the Arduino Nano, C++'s efficient memory management and minimal runtime overhead are essential. We can precisely control memory allocation and ensure optimal use of the available RAM. The language's support for namespaces, classes, and templates helps maintain clean code architecture despite the system's complexity. C++'s strong type system helps catch errors at compile-time rather than runtime, which is crucial for a system that needs to operate autonomously. Additionally, C++ allows us to create clean abstractions over hardware components while maintaining direct access when needed, making the code both maintainable and efficient. The object-oriented features facilitate code reuse and modular design, making it easier to extend or modify the robot's functionality.

The combination of Arduino Nano's capabilities with C++'s features allows us to create a sophisticated line following robot that not only performs its primary function but also provides valuable insights into its operation through advanced logging and analysis capabilities.

2 Line Following Robot Operation

The robot's operation is divided into two distinct configuration levels: build-time configuration through software compilation and runtime operation of RoboZero.

At build time, critical parameters are set through compilation flags and constants. The DEBUG_LEVEL flag determines the robot's operational mode: normal operation (0), analysis mode (1), or speed mode (2). Each mode compiles with different features and behaviors. Other build-time configurations include PID default constants, sensor weights, speed profiles, and timing parameters. These decisions fundamentally shape the robot's behavior and available features.

During runtime operation, the robot's functioning begins with a calibration phase, essential for adapting to varying lighting conditions and surface characteristics. Upon powering up, the robot waits for the first button press, which initiates the calibration sequence. During calibration, the robot samples each of its six line sensors multiple times, establishing minimum and maximum values for each sensor. These values create a baseline for converting raw sensor readings into meaningful position data.

After calibration, the robot waits for a second button press to begin its line following operation. The six-sensor array continuously reads the line position, with the outer sensors serving as markers for extreme deviations and the inner sensors providing precise positioning data. Each sensor reading is normalized using the calibration data and weighted based on its position in the array. The weighted average of these readings determines the robot's position relative to the line.

The control system operates on a continuous loop, processing sensor data and adjusting motor outputs. A PID (Proportional-Integral-Derivative) controller calculates the necessary corrections based on the line position error - the difference between the current position and the desired center position. The proportional component provides immediate response to position errors, while the derivative component helps predict and dampen oscillations, resulting in smooth motion.

Speed control is managed dynamically based on the current situation. In straight sections, the robot can maintain higher speeds, while curves require speed adjustment for stability. The system includes a boost mechanism that temporarily increases speed when exiting curves to optimize lap times. The operating mode, determined at build time, affects the available speed ranges and control parameters.

Course markers on the track trigger specific behaviors. The robot can detect intersections, lap markers, and mode-change markers using dedicated marker sensors. When passing over these markers, the robot adjusts its operation accordingly - counting laps, changing speeds, or preparing to stop. The system is designed to complete a configurable number of laps before automatically stopping.

If compiled with debugging enabled (DEBUG_LEVEL > 0), the logging system operates continuously during the robot's operation. In analysis mode, it captures detailed performance data including line position, error values, motor speeds, and PID corrections. This data is temporarily stored in circular buffers and written to flash memory when safe conditions are met - typically during straight sections where precise control is less critical. The system also records significant events such as marker detections, mode changes, and lap completions.

When the programmed number of laps is completed, the robot enters its stopping sequence. It gradually reduces speed to ensure controlled deceleration and precise stopping. After stopping, if debugging is enabled, the collected performance data remains stored in flash memory, ready for retrieval and analysis.

Data retrieval, available only in debug builds, occurs through a USB interface when the robot is stationary. Upon receiving the appropriate command through the serial interface, the robot transmits its stored data in a structured format, including session headers, performance records, event logs, and lap statistics. This data can then be analyzed to optimize the robot's parameters such as PID constants, speed profiles, and acceleration rates.

The entire system prioritizes real-time performance while ensuring data collection doesn't interfere with the robot's primary line-following function. Error checking and recovery mechanisms are implemented throughout the system, from sensor reading validation to data storage verification, ensuring reliable operation even under challenging conditions.

2.1 Operating Procedure

2.1.1 Initial Setup

1. Place the robot near the course
2. Power on the robot
3. Wait for initial setup delay (600ms)
    - Status LED will be on during this period
    - Motors will be inactive
4. After delay, LED turns off and robot is ready for calibration

2.1.2 Calibration Process

1. Press the start button for first calibration phase
    - LED will turn on
2. The calibration process:
    - Takes 400 samples from each sensor
    - 30ms delay between samples (total ~12 seconds)
    - Establishes minimum and maximum values for each sensor
3. After calibration completes:
    - LED turns off
    - Robot waits for second button press

2.1.3 Operation Start

1. Place robot on the track
2. Press start button again to begin operation
    - LED turns on
    - Robot starts line following operation
3. Initial operating parameters:
    - Speed mode begins at BASE_FAST (115)
    - PID control active with default parameters
    - Normal operating mode engaged

2.1.4 During Operation

The robot recognizes three marker patterns: 1. Finish line marker (both sensors) - Updates lap count - Triggers stop sequence on second detection 2. Speed mode marker (left sensor only) - Toggles between normal and precision mode - In precision mode: SPEED_SLOW - In normal mode: BASE_FAST 3. Intersection marker (both sensors) - Logged but no special action taken

2.1.5 Stop Sequence

The robot will stop automatically when: 1. Second finish line is detected 2. Stop sequence activates: - Speed reduces to SPEED_BRAKE - After 50ms deceleration - Final stop after 300ms - Motors power off

2.1.6 Data Retrieval (Debug Mode Only)

If DEBUG_LEVEL > 0: 1. When robot stops, flash memory is marked as ready 2. LED displays transmission pattern: - Alternates between slow blink (1000ms) and fast blink (300ms) - Pattern switches every 3000ms 3. Data can be retrieved through serial interface 4. After successful transmission: - Log ready flag is cleared - LED pattern stops

2.1.7 Error Recovery

If line is lost: 1. Robot uses last valid position 2. Position is forced to extreme (-100 or 100) based on last direction 3. PID controller attempts to recover 4. Robot continues operation if line is found

2.1.8 Operating Modes

Two base operating speeds: 1. Normal Mode (BASE_FAST): - Base speed of 115 - Curve speed reduction active - Boost after curves (if not in precision mode)

2. Precision Mode:
    - Activated by left marker
    - Uses SPEED_SLOW

- Disables boost feature
- More conservative operation

Debug Operating Modes (if DEBUG_LEVEL > 0): 1. Analysis Mode (DEBUG_LEVEL = 1): - 5 laps - Conservative speeds - Full data logging

2. Speed Mode (DEBUG_LEVEL = 2):

   - 3 laps
   - Maximum performance
   - Full data logging

3 RoboZero Modules Description

As shown in Figure 1, RoboZero's architecture is organized into distinct layers, each responsible for specific aspects of the robot's operation.

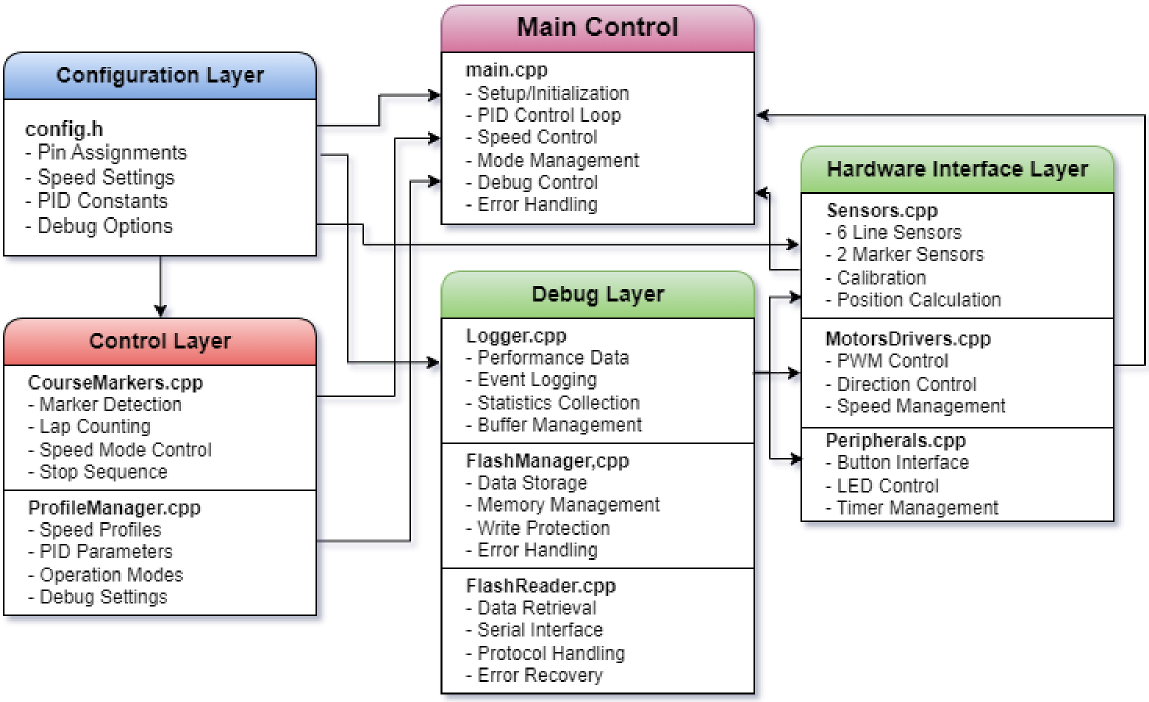

Figure 3.1: Figure 1: RobotZero's Block Diagram

Let's examine each module and its key features:

3.1 Configuration Layer

3.1.1 config.h

The configuration hub of the system, this module defines all crucial parameters including pin assignments, speed settings, and control constants. A notable feature is its use of conditional compilation (`#if DEBUG_LEVEL > 0`) to ensure zero overhead in normal operation mode, demonstrating our commitment to efficiency.

3.1.2 globals.h

Manages global state variables that need to be accessed across different modules. While global variables are generally discouraged, here they serve a crucial role in maintaining real-time performance by avoiding function call overhead for frequently accessed states.

3.1.3 debug.h

Implements a sophisticated debug message system that stores strings in Flash memory instead of RAM, using `PROGMEM` for optimal memory usage. This approach ensures that debug capabilities don't impact the robot's limited RAM resources.

3.2 Hardware Interface Layer

3.2.1 Sensors

Manages six line sensors and two marker sensors through a calibration-based approach. The unique feature here is the weighted average calculation that provides precise positional data. The system maintains both raw and processed values, enabling real-time adjustments while preserving original readings for analysis.

3.2.2 MotorsDrivers

Implements motor control using PWM, with a key feature being its ability to handle both forward and reverse motion through a single interface. The module includes built-in protection against invalid PWM values, ensuring safe operation even under software errors.

3.2.3 Peripherals

Handles external interfaces including button input and LED status indication. Notable is its debounce implementation that maintains responsiveness while ensuring reliable button detection, essential for both operation and calibration phases.

3.3 Control Layer

3.3.1 CourseMarkers

Processes track markers using a state machine approach to detect different patterns (finish line, speed mode changes, intersections). Its sophisticated detection system can differentiate between various marker combinations while maintaining reliable operation under varying light conditions.

3.3.2 ProfileManager

Manages different operation profiles (analysis and speed modes). The key innovation here is its transparent speed value translation system, which allows the same base code to operate under different performance parameters without modification.

3.4 Debug Layer

3.4.1 Logger

Implements a sophisticated logging system using circular buffers to maintain performance. A key feature is its ability to write to flash memory only during straight-line sections, ensuring logging doesn't interfere with critical control operations.

3.4.2 FlashManager

Handles flash memory operations with built-in error checking and recovery mechanisms. Notable is its page-aligned writing system that maximizes flash memory lifespan while ensuring data integrity.

3.4.3 FlashReader

Manages data retrieval through a structured protocol, including checksums for data validation. The module implements a multi-marker system to ensure reliable data transmission even under noisy serial connections.

3.5 Main Control

3.5.1 main.cpp

The core control loop implementing PID-based line following. A significant feature is its non-blocking setup sequence that maintains system responsiveness during initialization and calibration. The module seamlessly integrates debug features when compiled with DEBUG_LEVEL > 0 while maintaining optimal performance in normal operation.

4 Configuration Layer

The Configuration Layer serves as the foundation of the RobotZero system, providing centralized control over all system parameters. This layer plays a crucial role in both development and runtime operation, allowing fine-tuning of robot behaviour without modifying core logic. Its design prioritizes both flexibility and efficiency, using compile-time constants to ensure zero runtime overhead while maintaining high configurability.

4.1 Configuration Values (config.h)

The config.h file is structured into logical sections, each handling specific aspects of the robot's configuration. Let's examine each section in detail:

4.1.1 Debug Configuration

This section controls the debugging features of the system. The `DEBUG_LEVEL` setting determines the robot's operating mode and what features are compiled into the final binary.

```
// Set to 1 for analysis mode, 2 for speed mode, or 0 for normal operation
#ifndef DEBUG_LEVEL
#define DEBUG_LEVEL 0
#endif

#if DEBUG_LEVEL > 0
#include "DataStructures.h"  // Include debug-related structures

// Debug configuration
static constexpr uint8_t DEBUG_LAPS_MODE1 = 5;    // Analysis mode laps
static constexpr uint8_t DEBUG_LAPS_MODE2 = 3;    // Speed mode laps

// Logging parameters
static constexpr uint16_t SAMPLE_RATE_STRAIGHT = 50;   // Sampling in straight lines
static constexpr uint16_t SAMPLE_RATE_CURVE = 20;      // Sampling in curves
static constexpr uint16_t LOG_BUFFER_SIZE = 64;        // Circular buffer size

// Flash memory parameters
static constexpr uint32_t FLASH_LOG_START = 0x1000;    // Log start address
static constexpr uint16_t FLASH_PAGE_SIZE = 256;       // Flash page size
static constexpr uint32_t FLASH_CONTROL_BYTE = 0x0800; // Control byte location
static constexpr uint8_t FLASH_LOG_READY = 0xAA;       // Log ready indicator
#endif
```

The debug configuration implements a conditional compilation system that ensures optimal performance in normal operation. When `DEBUG_LEVEL` is set to `0`, all debugging code is completely excluded from the final binary, resulting in no runtime overhead. Setting `DEBUG_LEVEL` to `1` activates the analysis mode, where the robot operates at moderate speeds and collects comprehensive data about its performance, including position errors, motor speeds, and **PID corrections**, completing `5` laps for detailed analysis. In speed mode, activated with `DEBUG_LEVEL 2`, the robot performs `3` laps at maximum speed while still collecting performance data, allowing for optimization of high-speed behaviour. The system adjusts its sampling rate based on track conditions - sampling more frequently in curves where behaviour is more dynamic, and at a lower rate in straight sections to conserve memory. All collected data is stored in a structured format in flash memory, organized to maximize data integrity and facilitate post-run analysis.

The flash memory organization is carefully structured to maximize efficiency and reliability. The logging system begins storing data at address `0x1000` (`FLASH_LOG_START`), providing ample space for system data in lower memory addresses. Each page of flash memory is `256` bytes (`FLASH_PAGE_SIZE`), allowing efficient writing operations that balance between memory usage and write cycles. A control byte located at address 0x0800 (`FLASH_CONTROL_BYTE`) serves as a state indicator for the logging system. When this byte contains the value 0xAA (`FLASH_LOG_READY`), it signals that valid performance data is available for retrieval, ensuring proper synchronization between data logging and retrieval operations.

4.1.2 Pin Configuration

Defines all hardware connections, centralizing pin assignments for easy modification and hardware revision control.

```
// Only modify if changing physical robot connections
static const uint8_t PIN_START_BUTTON = 11;      // Start/calibrate button
static const uint8_t PIN_STATUS_LED = 13;        // Status LED
static const uint8_t PIN_MOTOR_LEFT_FWD = 7;     // Left Motor Forward
static const uint8_t PIN_MOTOR_LEFT_REV = 4;     // Left Motor Reverse
static const uint8_t PIN_MOTOR_LEFT_PWM = 3;     // Left Motor Speed
static const uint8_t PIN_MOTOR_RIGHT_FWD = 8;    // Right Motor Forward
static const uint8_t PIN_MOTOR_RIGHT_REV = 9;    // Right Motor Reverse
static const uint8_t PIN_MOTOR_RIGHT_PWM = 10;   // Right Motor Speed

// Sensor pins
static const uint8_t PIN_LINE_LEFT_EDGE = A6;    // Leftmost sensor
static const uint8_t PIN_LINE_LEFT_MID = A5;
static const uint8_t PIN_LINE_CENTER_LEFT = A4;
static const uint8_t PIN_LINE_CENTER_RIGHT = A3;
static const uint8_t PIN_LINE_RIGHT_MID = A2;
static const uint8_t PIN_LINE_RIGHT_EDGE = A1;   // Rightmost sensor
static const uint8_t PIN_MARKER_LEFT = A7;       // Left marker
static const uint8_t PIN_MARKER_RIGHT = A0;      // Right marker
```

The pin configuration employs a systematic approach to hardware interface management. Each pin assignment is thoroughly documented with clear comments indicating its purpose, from motor control signals to sensor inputs, making hardware modifications and debugging straightforward. Related pins are logically grouped together - motor control pins are clustered by function (forward, reverse, and PWM for each motor), while sensor pins are arranged according to their physical layout on the robot (from left edge to right edge). The use of static const declarations for pin assignments not only makes the code more readable but also allows the compiler to optimize memory usage by storing these values in program memory rather than RAM. This approach maintains flexibility for hardware modifications while ensuring efficient runtime performance, as these values are resolved at compile time rather than being calculated during program execution.

4.1.3 Speed Parameters

Defines the various speed levels used by the robot, providing a comprehensive speed control system.

```
// Base speeds - do not modify without thorough testing
static constexpr uint8_t SPEED_STOP = 0;        // Stopped
static constexpr uint8_t SPEED_STARTUP = 80;    // Initial movement
static constexpr uint8_t SPEED_TURN = 100;      // Turn speed
static constexpr uint8_t SPEED_BRAKE = 120;     // Braking speed
static constexpr uint8_t SPEED_CRUISE = 140;    // Medium speed
static constexpr uint8_t SPEED_SLOW = 160;      // Precision mode
static constexpr uint8_t SPEED_FAST = 180;      // High speed
static constexpr uint8_t SPEED_BOOST = 200;     // Boost speed
static constexpr uint8_t SPEED_MAX = 220;       // Maximum speed

// Speed control parameters
static constexpr uint8_t ACCELERATION_STEP = 25;   // Speed increase step
static constexpr uint8_t BRAKE_STEP = 60;         // Speed decrease step
static constexpr uint8_t TURN_SPEED = 120;        // Curve speed
static constexpr uint8_t TURN_THRESHOLD = 45;     // Curve detection
static constexpr uint8_t STRAIGHT_THRESHOLD = 20;  // Straight line detection
static constexpr uint8_t BOOST_DURATION = 10;     // Boost time
static constexpr uint8_t BOOST_INCREMENT = 20;    // Boost step
```

These speed constants and control parameters are extensively used throughout the codebase. The CourseMarkers class uses them to determine appropriate speeds for different track sections, with `TURN_THRESHOLD` and `STRAIGHT_THRESHOLD` helping identify track geometry. In the main control loop, these values drive the PID controller's response, with `ACCELERATION_STEP` and `BRAKE_STEP` ensuring smooth speed transitions. When `DEBUG_LEVEL` is greater than `0`, the *ProfileManager* modifies these base values according to the current operating mode, allowing for different performance profiles while maintaining the same core control logic.

4.1.4 Control Parameters

Defines the PID controller and sensor processing parameters.

```
// PID Control Parameters
static constexpr float K_PROPORTIONAL_DEFAULT = 5.0f;
static constexpr float K_DERIVATIVE_DEFAULT = 600.0f;
```

```
static constexpr float FILTER_COEFFICIENT_DEFAULT = 0.6f;

// Sensor Parameters
static const uint8_t NUM_SENSORES = 6;
static constexpr int16_t SENSOR_MAX_VALUE = 1023;
static constexpr int16_t SENSOR_MIN_VALUE = 0;
static constexpr int16_t SENSOR_THRESHOLD = 120;

// Sensor Weights
static constexpr float SENSOR_WEIGHT_S1 = -2.5f;  // Far left
static constexpr float SENSOR_WEIGHT_S2 = -1.2f;  // Left
static constexpr float SENSOR_WEIGHT_S3 = -0.6f;  // Center-left
static constexpr float SENSOR_WEIGHT_S4 = 0.6f;   // Center-right
static constexpr float SENSOR_WEIGHT_S5 = 1.2f;   // Right
static constexpr float SENSOR_WEIGHT_S6 = 2.5f;   // Far right
```

The control system parameters represent the core of the robot's line-following behaviour. The PID controller uses carefully tuned constants, with a proportional gain (`K_PROPORTIONAL_DEFAULT`) of `5.0` providing immediate response to position errors, while the high derivative gain (`K_DERIVATIVE_DEFAULT`) of `600.0` helps predict and dampen oscillations. A filter coefficient of `0.6` balances between noise reduction and response time in the derivative calculation.

The sensor array consists of six sensors (`NUM_SENSORES`), each providing analog readings from `0` to `1023` (`SENSOR_MIN_VALUE` to `SENSOR_MAX_VALUE`). A threshold value of `120` helps distinguish between line and background surface conditions. The sensor weights are particularly crucial, implementing a distributed sensing system where outer sensors (`±2.5`) have greater influence than inner ones (`±0.6`), creating a non-linear response that enhances stability in straight lines while maintaining sensitivity to curves. These weights are asymmetrical around the center point, allowing the robot to detect and respond to position changes with increasing urgency as it deviates further from the line. When processed together in the main control loop, these parameters enable the robot to maintain precise line following while adapting to various track conditions and geometries.

4.1.5 Timing Parameters

Controls various timing aspects of the robot's operation.

```
// Delays and Timings
static const uint16_t SETUP_DELAY = 600;            // Initial setup
static const uint16_t CALIBRATION_SAMPLES = 400;    // Calibration precision
static const uint8_t CALIBRATION_DELAY = 30;        // Sample interval
static const uint16_t STOP_DELAY = 300;             // Final stop timing
static const uint16_t DEBOUNCE_DELAY = 50;          // Button debounce
```

The timing parameters, all specified in milliseconds, govern critical operational sequences of the robot. The `SETUP_DELAY` of `600ms` allows the system to stabilize after power-up, ensuring all sensors and components are ready for operation. During calibration, the system takes `400` samples (`CALIBRATION_SAMPLES`) from each sensor, with a `30ms` interval (`CALIBRATION_DELAY`) between samples, providing a comprehensive baseline for sensor readings while balancing between accuracy and calibration time - resulting in a total calibration period of approximately `12 seconds`. When the robot detects its final lap marker, it initiates a controlled stop sequence lasting `300ms` (`STOP_DELAY`), allowing for smooth deceleration. Button inputs are processed with a `50ms` debounce period (`DEBOUNCE_DELAY`), effectively filtering out mechanical noise and contact bounce while maintaining responsive user interaction.

These timing values were established during the November 2024 competition where RoboZero secured second place, proving their effectiveness under competitive conditions.

4.2 Global Variables Management (globals.h)

The `globals.h` file represents a strategic decision in RobotZero's architecture, implementing a carefully selected set of global variables that require system-wide access. While global variables are generally discouraged in software development, their use here is justified by the real-time nature of the system and the Arduino Nano's limited resources.

```cpp
#ifndef GLOBALS_H
#define GLOBALS_H

// Global control variables
extern int currentSpeed;            // Base speed
extern bool isRobotStopped;         // Robot stopped state
extern bool isStopSequenceActive;   // Stopping sequence active
extern int lapCount;                // End marker counter
extern bool isPrecisionMode;        // Slow mode active

#endif // GLOBALS_H
```

The `currentSpeed` variable serves as the base speed reference for the entire system. It is modified by various components including the `CourseMarkers` class during turns, the main control loop during PID corrections, and the `ProfileManager` when operating in debug modes. By maintaining this as a global variable, we avoid the overhead of function calls and parameter passing in time-critical control loops.

The robot's state is tracked through three critical boolean flags. `isRobotStopped` indicates when the robot has completed its run or encountered a stop condition, allowing all components to safely cease operations. `isStopSequenceActive` manages the controlled deceleration process, triggered when the robot reaches its final lap, ensuring smooth and precise stopping. The `isPrecisionMode` flag enables the system to switch between normal and precision operation modes, affecting speed calculations and control parameters throughout the system.

The `lapCount` variable keeps track of completed laps, crucial for both normal operation and debugging modes. In normal mode, it triggers the stop sequence after one lap, while in debug modes (controlled by `DEBUG_LEVEL`), it follows the specified number of laps (`5` for analysis mode, `3` for speed mode).

All these variables are declared as external (`extern`) in the header file, with their actual definitions residing in `main.cpp`. This approach maintains proper encapsulation while allowing necessary access across the system. Each variable is initialized at system startup and modified only in specific, well-defined circumstances, ensuring predictable behaviour despite their global nature.

4.3 Debug System Configuration (debug.h)

The `debug.h` file implements an efficient debugging system that provides comprehensive diagnostic information without compromising the robot's performance. The system's most notable feature is its use of **Flash memory for string storage**, preserving valuable RAM for critical operations.

```
#ifndef DEBUG_H
#define DEBUG_H

#include "config.h"
#include <avr/pgmspace.h>

// Store debug messages in Flash memory instead of RAM
const char DEBUG_BASE[] PROGMEM = "Base: ";
const char DEBUG_ERROR[] PROGMEM = " Error: ";
const char DEBUG_CORRECTION[] PROGMEM = " Correction: ";
const char DEBUG_GEOMETRY[] PROGMEM = "Geometry: ";
const char DEBUG_RIGHT_MARKER[] PROGMEM = "rightMarkerDetected: ";
const char DEBUG_LEFT_MARKER[] PROGMEM = " leftMarkerDetected: ";
const char DEBUG_A0[] PROGMEM = " A0: ";
const char DEBUG_A7[] PROGMEM = " A7: ";
const char DEBUG_SLOW_MODE[] PROGMEM = "Slow mode activated";
const char DEBUG_FAST_MODE[] PROGMEM = "Fast mode activated";
const char DEBUG_INTERSECTION[] PROGMEM = "Intersection detected";
const char DEBUG_SETUP_START[] PROGMEM = "Starting setup";
const char DEBUG_SETUP_COMPLETE[] PROGMEM = "Setup completed";
```

These string constants are stored in program memory using the `PROGMEM` attribute. This approach saves precious RAM space on the Arduino Nano, which only has `2KB` available. To facilitate reading these stored strings, the system implements a helper function:

```
// Helper function to print strings from Flash
inline void debugPrintFlash(const char* str) {
    char c;
    while ((c = pgm_read_byte(str++))) {
        Serial.write(c);
    }
}
```

The system provides a set of debugging macros that are completely eliminated when debugging is disabled. These macros are defined based on the `DEBUG_LEVEL` configuration:

```
// Debug macros - only active when DEBUG_LEVEL > 0
#if DEBUG_LEVEL > 0
#define DEBUG_BEGIN(x) Serial.begin(x)
#define DEBUG_PRINT(x) debugPrintFlash(x)
#define DEBUG_PRINTLN(x) do { debugPrintFlash(x); Serial.println(); } while(0)
#define DEBUG_PRINT_VAL(x) Serial.print(x)
#define DEBUG_PRINTLN_VAL(x) Serial.println(x)
```

```
#else
#define DEBUG_BEGIN(x)
#define DEBUG_PRINT(x)
#define DEBUG_PRINTLN(x)
#define DEBUG_PRINT_VAL(x)
#define DEBUG_PRINTLN_VAL(x)
#endif
```

When DEBUG_LEVEL is `0`, these macros expand to nothing, ensuring zero overhead in the compiled code. When debugging is enabled, they provide different printing capabilities: `DEBUG_PRINT` and `DEBUG_PRINTLN` handle Flash-stored strings, while `DEBUG_PRINT_VAL` and `DEBUG_PRINTLN_VAL` handle direct value output. The do-while construct in `DEBUG_PRINTLN` ensures proper behaviour when the macro is used in if-else statements.

In practice, these macros are used throughout the codebase to provide diagnostic information. For example, during sensor readings:

```
DEBUG_PRINT("rightMarkerDetected: ");
DEBUG_PRINT_VAL(rightMarkerDetected);
DEBUG_PRINT(" leftMarkerDetected: ");
DEBUG_PRINT_VAL(leftMarkerDetected);
DEBUG_PRINTLN("");
```

The system outputs debug information at `115200` baud when enabled, allowing real-time monitoring of the robot's behaviour while maintaining efficient execution.