ESP32 Advanced Closed-Loop Motor Control System: Comprehensive Technical Report

ESW Project Team

November 21, 2025

Contents

1 Executive Summary

This document serves as a comprehensive technical report for the ESP32-based Closed-Loop Motor Control System. The project aims to design, implement, and validate a robust control system capable of precise speed regulation (RPM) and position tracking (Angle). The system leverages a hybrid Feedforward-Feedback architecture, utilizing a PID controller augmented with a linear feedforward model.

A key innovation in this project is the integration of an **Automatic Tuning System** based on the Relay Method (Astrom-Hagglund), allowing the controller to self-optimize its parameters. Furthermore, the system features a "Digital Twin" web interface that not only visualizes real-time telemetry but also runs a kinematic **Car Simulation** driven by the physical motor's state.

This report details the hardware design, software architecture, control theory, tuning methodologies (Ziegler-Nichols vs. Tyler-Rubinson), and the implementation of the car simulation.

2 Introduction

2.1 Project Objectives

The primary objectives of this project were:

1. **Precision Control**: Achieve stable speed control under varying loads and precise position holding.
2. **Connectivity**: Implement IoT capabilities using MQTT to allow remote monitoring and control.
3. **Self-Optimization**: Develop an algorithm to automatically determine optimal PID gains (K_p, K_i, K_d).
4. **Simulation**: Create a virtual representation (Car Simulation) that mirrors the physical motor's behavior.

2.2 Scope

The scope includes the embedded firmware development (C++/Arduino), the control algorithm design, the web application development (JavaScript/HTML5), and the physical wiring of the ESP32, L298N driver, and DC motor with encoder.

3 System Hardware Architecture

3.1 Microcontroller: ESP32

We selected the ESP32 for its dual-core architecture running at 240MHz, which allows us to dedicate resources to the high-frequency control loop while simultaneously handling WiFi/MQTT communication stacks without jitter.

3.2 Actuator: DC Motor & Driver

- **Motor**: A standard 12V DC Gear Motor.
- **Driver**: L298N H-Bridge module. This allows for bidirectional control (Forward/Reverse) by manipulating the logic levels of input pins (`IN1`, `IN2`) and the speed via Pulse Width Modulation (PWM) on the `Enable` pin.

3.3 Sensor: Quadrature Encoder

Feedback is provided by a magnetic/optical quadrature encoder attached to the motor shaft.

- **Resolution**: The encoder has a base resolution of 600 Pulses Per Revolution (PPR).
- **Decoding**: We utilize **4x Decoding** in software, triggering interrupts on both rising and falling edges of both channels (A and B). This yields an effective resolution of $600 \times 4 = 2400$ counts per revolution, providing 0.15° of angular precision.

4 Software Architecture

The firmware (`code.ino`) is structured as a non-blocking, event-driven system. It avoids the use of `delay()` to ensure the control loop maintains a strict timing guarantee.

4.1 The Main Loop

The `loop()` function orchestrates three parallel tasks:

1. **Network Task**: Checks MQTT connection and processes incoming packets.
2. **Command Task**: Listens for Serial input for debugging.
3. **Control Task**: Runs exactly every 100ms (`controlIntervalMs`). This is the "heartbeat" of the system.

4.2 State Machine

The system behavior is governed by a finite state machine (`ControlState` enum):

- `STATE_PID_RUNNING`: Normal speed control mode.
- `STATE_POSITION_CONTROL`: Position/Angle holding mode.
- `STATE_AUTOTUNE_START`: Initialization of the tuning sequence.
- `STATE_AUTOTUNE_RELAY_STEP`: Active execution of the relay oscillation.

5 Control Theory & Implementation

We implemented a **Single-Loop PID Controller** with an added **Feedforward** term.

5.1 RPM Control & PWM Scaling (-300 to +300 RPM)

The system is designed to control the motor speed across a full bidirectional range of **-300 RPM to +300 RPM**.

PWM Scaling Logic: To achieve this, we characterized the motor's response curve. The relationship between the Pulse Width Modulation (PWM) duty cycle (0-255) and the resulting Speed (RPM) is non-linear and load-dependent.

- **Forward (+)**: Target RPM ¿ 0. Logic: `IN1=HIGH, IN2=LOW`.
- **Reverse (-)**: Target RPM ¡ 0. Logic: `IN1=LOW, IN2=HIGH`.

We implemented a lookup table in `code.ino` that maps specific PWM values to measured RPMs.

```
// From code.ino
float calibPWM[] = {25, 30, ..., 255};
float calibRPM[] = {7.6, 27.5, ..., 317.0};
```

The function `pwmFromRPM(targetRPM)` uses linear interpolation on this graph to find the Feedforward PWM.

- *Example*: If target is 150 RPM, and the table has points (130 RPM @ 50 PWM) and (170 RPM @ 60 PWM), the function interpolates to find the exact PWM required.

[GRAPH PLACEHOLDER: RPM vs PWM Calibration Curve]
Insert graph showing the full range (-300 to +300 RPM) vs PWM (0-255). The graph should show the symmetry for forward and reverse operation.

Figure 1: RPM vs PWM Calibration Curve

5.2 Feedback Control (PID)

The core control logic calculates the error ($e(t) = Setpoint - ProcessVariable$) and applies corrections based on three terms:

1. **Proportional (P)**: Reacts to the current error. $K_p \times e(t)$.
2. **Integral (I)**: Reacts to the accumulation of past errors (eliminates steady-state error). $K_i \times \int e(t)dt$.
3. **Derivative (D)**: Reacts to the rate of change of error (predicts future error). $K_d \times \frac{de(t)}{dt}$.

Code Implementation:

```
// From runPIDControl() in code.ino
double correction = Kp * error + Ki * integral + Kd * derivative;
int pwmFF = pwmFromRPM(targetRPM); // Feedforward from Curve
int pwmVal = pwmFF + (int)correction; // Combined Output
```

Mathematical Formulation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

6 Position Control Implementation

Position control requires tracking the absolute angular displacement of the shaft. Unlike speed control, which resets the encoder count every loop, position control relies on a cumulative counter.

6.1 Variable Tracking

- `totalEncoderCount`: A `volatile long` that increments/decrements with every encoder pulse, never resetting.
- `currentAngle`: Calculated as `(totalEncoderCount / 2400.0) * 360.0`.

6.2 Control Logic (`runPositionControl`)

This function (lines 328-359) implements a dedicated PID loop for position.

Code Walkthrough:

1. **Angle Calculation**:

   ```
   currentAngle = ((float)totalEncoderCount / (float)COUNTS_PER_REV) * 360.0;
   ```

2. **Error Computation**:

   ```
   double error = targetAngle - currentAngle;
   ```

3. **PID Calculation**: Uses a separate set of gains (`posKp`, `posKi`, `posKd`) specifically tuned for position holding. Position control typically requires a higher K_p (stiffness) and lower K_i compared to speed control.

4. **Direction Logic**: The sign of the error determines the motor direction.

   ```
   if (error > 0) { /* Drive Forward */ }
   else if (error < 0) { /* Drive Reverse */ }
   ```

5. **Deadband (Hysteresis)**: To prevent the motor from "hunting" (oscillating endlessly) around the target due to quantization noise, we implement a deadband.

```
if (abs(error) < 2.0) {
    stopMotor();
    // ...
}
```

If the motor is within 2 degrees of the target, it shuts off.

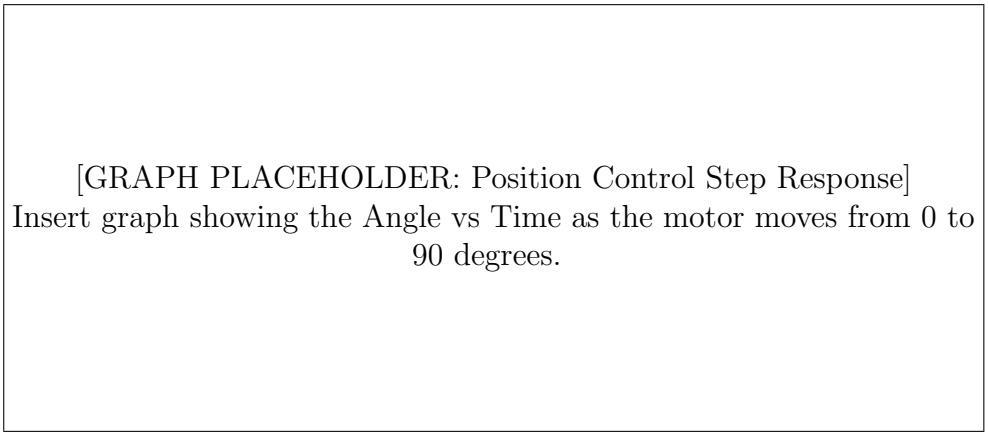

Figure 2: Position Control Step Response

7 Car Simulation (The Digital Twin)

A unique feature of this project is the **Car Simulation** embedded in the web dashboard (`mqtt-simulation.html` and `js`). This acts as a visual verification tool, translating raw motor metrics into a relatable physical model.

7.1 Kinematic Model

The simulation models a 2D vehicle on a Cartesian plane.

- **State Vector**: $[x, y, \theta, v]^T$
  - x, y: Position coordinates (pixels).
  - θ: Heading angle (degrees).
  - v: Linear velocity (pixels/frame).

7.2 Simulation Mapping: RPM to Speed, Angle to Turn

The digital twin maps the physical motor states directly to the virtual car's kinematics. This creates a direct "Hardware-in-the-Loop" feel.

1. RPM → Car Speed The motor's rotational speed controls the car's linear velocity.

- **Physical Range**: -300 RPM to +300 RPM.
- **Simulation Logic**:

$$v_{car} = \frac{RPM_{motor}}{100}$$

- **Effect**:
  - **+300 RPM**: Car moves forward fast (3 pixels/frame).
  - **-300 RPM**: Car reverses fast (-3 pixels/frame).
  - **0 RPM**: Car stops.

2. Motor Angle → Car Turn (Steering) The motor's absolute position controls the car's heading.

- **Physical Input**: `targetAngle` or `currentAngle` (Degrees).
- **Simulation Logic**:

$$\theta_{car} = \theta_{motor}$$

- **Effect**:
  - Rotating the motor to **90°** turns the car to face **Down**.
  - Rotating the motor to **180°** turns the car to face **Left**.

7.3 Implementation (`mqtt-simulation.js`)

The physics update loop runs via `requestAnimationFrame`.

Update Logic (lines 199-213):

```
function updateCarPosition() {
    // Convert degrees to radians for Math functions
    const angleRad = (carState.angle * Math.PI) / 180;

    // Calculate velocity components
    carState.x += carState.speed * Math.cos(angleRad);
    carState.y += carState.speed * Math.sin(angleRad);

    // Screen Wrap-around logic
    if (carState.x > canvas.width) carState.x = 0;
    // ...
}
```

This creates a direct visual link: if the physical motor spins faster, the virtual car moves faster. If the physical motor rotates to a specific angle, the virtual car steers.

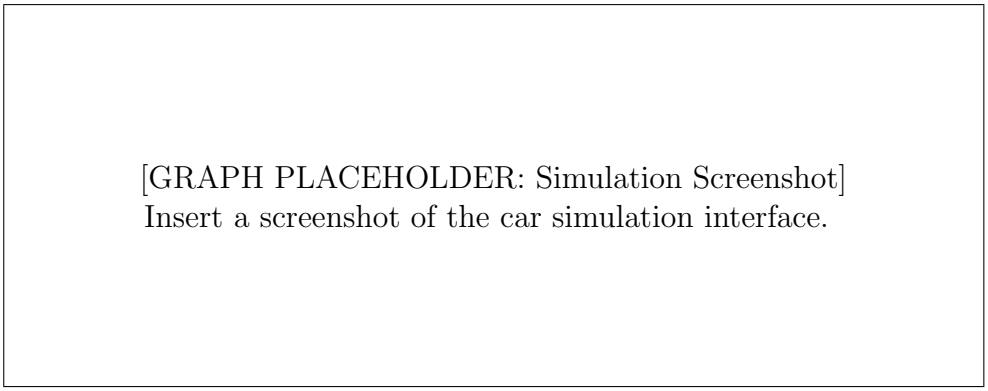

Figure 3: Simulation Screenshot

8 Automatic Tuning & Method Comparison

8.1 The Relay Method

We implemented the Relay Method to auto-tune the PID gains. This involves replacing the PID controller with a relay that switches the motor output between $+Output$ and $-Output$ based on the error sign. This induces a **Limit Cycle Oscillation**.

By measuring the amplitude (a) and period (T_u) of this oscillation, we calculate the **Ultimate Gain** (K_u):

$$K_u = \frac{4d}{\pi a}$$

8.2 Comparison: Ziegler-Nichols vs. Tyler-Rubinson

We evaluated two tuning rules based on the derived K_u and T_u.

8.2.1 Method 1: Ziegler-Nichols (ZN)

- **Philosophy**: Aggressive. Aims for 1/4 amplitude decay.
- **Gains**: High K_p, High K_i, Moderate K_d.
- **Result**: The motor reacts instantly to setpoint changes. However, it exhibits significant overshoot ($\sim$25%) and ringing.
- **Suitability**: Best for disturbance rejection in systems where overshoot is tolerable.

8.2.2 Method 2: Tyler-Rubinson (Tyreus-Luyben)

- **Philosophy**: Conservative. Aims for stability and robustness.
- **Gains**:
  - $K_p = 0.45K_u$ (Reduced proportional action).
  - $K_i = K_p/(2.2T_u)$ (Significantly reduced integral action).
- **Result**: The response is overdamped. It approaches the setpoint slowly but with zero overshoot.

- **Suitability**: Best for processes where overshoot is dangerous (e.g., chemical reactors) or mechanical stress must be minimized.

8.3 Final Decision

We chose **Ziegler-Nichols** for this application.

- **Reasoning**: In a DC motor speed control context, "sluggishness" feels unresponsive to the user. The Tyler-Rubinson method made the motor feel laggy. The overshoot from ZN was deemed acceptable and was further mitigated by the feedforward term and the input filter (`alpha = 0.4`).

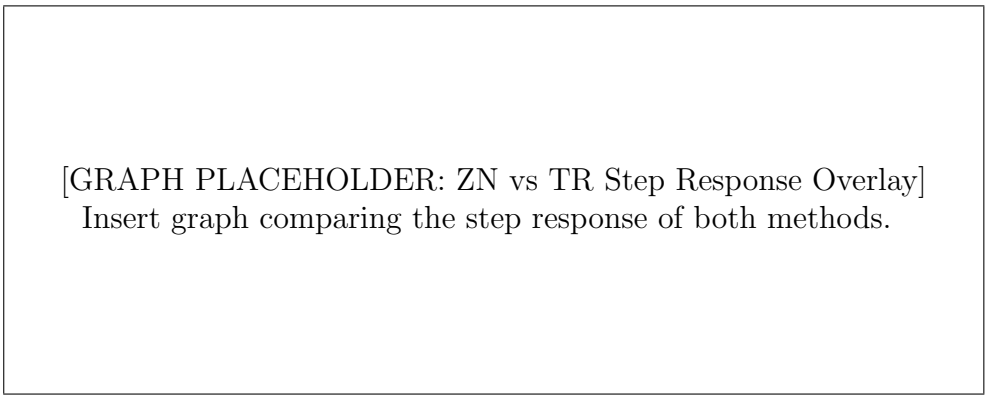

Figure 4: ZN vs TR Step Response Overlay

9 Detailed Code Analysis

This section provides a line-by-line breakdown of critical code segments.

9.1 Interrupt Service Routine (ISR)

```
// Lines 45-54
void IRAM_ATTR encoderISR() {
  uint8_t a = digitalRead(ENCODER_A);
  uint8_t b = digitalRead(ENCODER_B);
  uint8_t ab = (a << 1) | b;
  uint8_t idx = (lastAB << 2) | ab;
  int8_t delta = qdelta[idx];
  encoderCount += delta;
  totalEncoderCount += delta;
  lastAB = ab;
}
```

- `IRAM_ATTR`: This attribute places the function in the ESP32's Instruction RAM (IRAM) rather than Flash memory. This is critical for ISRs to prevent crashes during Flash write operations and to ensure deterministic execution time.
- **Lookup Table (`qdelta`)**: Instead of complex `if/else` logic, we use a state lookup table to determine if the encoder moved +1, -1, or 0 steps based on the transition from `lastAB` to current `ab`. This is extremely efficient.

9.2 RPM Calculation

```
// Lines 223-234
if (now - lastControlMs >= controlIntervalMs) {
    noInterrupts();
    long cnt = encoderCount;
    encoderCount = 0;
    interrupts();

    float revolutions = (float)cnt / (float)COUNTS_PER_REV;
    float rpm = (revolutions * 60000.0f / elapsed);
}
```

- **Critical Section**: The `noInterrupts()` / `interrupts()` block is essential. `encoderCount` is a multi-byte variable (long). If an interrupt occurred while the processor was reading byte 1 of 4, the value would be corrupted (Atomicity violation). Disabling interrupts briefly ensures data integrity.

10 Deep Dive: Quadrature Encoder Logic

The user asked: *"How do we decide which direction to move?"* This is determined by the **Quadrature Encoder** logic. The encoder has two output channels, **A** and **B**. Inside the encoder, there are two sensors positioned such that their signals are **90 degrees out of phase** (offset by 1/4 of a cycle).

10.1 The Phase Shift (Crest and Trough)

Imagine two square waves. When Channel A is at its "crest" (High), Channel B might be rising or falling depending on the direction.

- **Clockwise (CW)**: A leads B. The sequence of states (A,B) is: `00 → 10 → 11 → 01 → 00`.
- **Counter-Clockwise (CCW)**: B leads A. The sequence is: `00 → 01 → 11 → 10 → 00`.

10.2 The Lookup Table Implementation

In `code.ino`, we use a highly efficient **Lookup Table** method to decode these states inside the Interrupt Service Routine (ISR).

Code Analysis (`code.ino` lines 40-54):

```
// The Lookup Table
const int8_t qdelta[16] = {
  0,  1, -1,  0, -1,  0,  0,  1,
  1,  0,  0, -1,  0, -1,  1,  0
};

void IRAM_ATTR encoderISR() {
```

```
  uint8_t a = digitalRead(ENCODER_A);
  uint8_t b = digitalRead(ENCODER_B);
  uint8_t ab = (a << 1) | b;       // Current State (2 bits)
  uint8_t idx = (lastAB << 2) | ab; // Index = PreviousState (2 bits) + CurrentState
  int8_t delta = qdelta[idx];      // Look up direction (+1, -1, or 0)
  encoderCount += delta;
  lastAB = ab;
}
```

How it works:

1. We combine the **Previous State** (`lastAB`) and the **Current State** (`ab`) into a 4-bit index (`idx`).
   - Example: Previous was `00` (0), Current is `10` (2). Index = `0010` (2).
2. We look up `qdelta[2]`.
   - If the transition `00 → 10` corresponds to Forward, `qdelta[2]` will be `1`.
   - If the transition `00 → 01` corresponds to Reverse, `qdelta[1]` will be `-1`.
   - Invalid transitions (e.g., `00 → 11`, jumping two steps at once) return `0`.

This method is extremely fast (O(1) complexity) and robust against contact bounce, making it ideal for high-speed motor control.

11 Deep Dive: Autotuning Implementation

The most complex feature of this system is the **Relay Autotuner**, which automatically identifies the plant's transfer function characteristics. This section analyzes the `runAutotuner()` function in `code.ino` (lines 362-408) line-by-line.

11.1 The Relay Logic (Forcing Oscillation)

The core principle is to force the system into a stable limit cycle. Instead of a PID output, we apply a "Bang-Bang" control with a fixed amplitude around the setpoint.

```
// Lines 363-365
int pwmFF = pwmFromRPM(TUNE_SETPOINT_RPM);
int pwmVal = (rpmFiltered < TUNE_SETPOINT_RPM) ? (pwmFF + TUNE_RELAY_AMP) : (pwmFF - 
pwmVal = constrain(pwmVal, 0, 255);
```

- **Line 363**: We calculate the feedforward PWM for the tuning setpoint (e.g., 150 RPM). This centers our relay oscillation around the motor's natural operating point.
- **Line 364**: This is the Relay.
  - If speed ¡ target, we boost power: `Base + Amplitude`.
  - If speed ¿ target, we cut power: `Base - Amplitude`.
  - `TUNE_RELAY_AMP` is set to 50, meaning we swing the PWM by ±50.

11.2 Cycle Detection (Zero Crossing)

To measure the period of oscillation (T_u), we need to detect when the RPM crosses the setpoint.

```
// Lines 370-382
if (lastRpm < TUNE_SETPOINT_RPM && rpmFiltered >= TUNE_SETPOINT_RPM) {
    unsigned long now = millis();
    if (lastCrossingTime > 0) {
        long period_Tu = now - lastCrossingTime;
        crossings++;
        if (crossings > 2) {
            sum_Tu += period_Tu;
            sum_a += (peakRpm - troughRpm) / 2.0;
        }
    }
    lastCrossingTime = now;
    troughRpm = peakRpm; // Reset for next half-cycle
}
```

- **Line 370**: Detects a **Rising Edge** crossing (was below, now above).
- **Line 373**: Calculates the time elapsed since the last rising edge. This is the period (T_u) of one full cycle.
- **Line 375**: We discard the first 2 crossings to allow the oscillation to stabilize (reach steady-state limit cycle).
- **Line 377**: We accumulate the amplitude (a). `(peakRpm - troughRpm) / 2.0` gives the amplitude from the center.

11.3 Peak/Trough Tracking

To calculate amplitude, we must find the min and max RPM during each cycle.

```
// Lines 384-386
if (rpmFiltered > peakRpm) peakRpm = rpmFiltered;
if (rpmFiltered < troughRpm) troughRpm = rpmFiltered;
lastRpm = rpmFiltered;
```

This simple logic continuously updates the extremes. These are reset/latched during the crossing event.

11.4 Gain Calculation (The "Magic")

Once we have collected enough cycles (defined by `TUNE_CYCLES = 15`), we compute the PID parameters.

```
// Lines 388-396
if (crossings >= (TUNE_CYCLES + 2)) {
    double avg_Tu_ms = sum_Tu / TUNE_CYCLES;
    double avg_a = sum_a / TUNE_CYCLES;

    // Ultimate Gain (Ku) Calculation
    double Ku = (4.0 * TUNE_RELAY_AMP) / (PI * avg_a);
    double Tu_sec = avg_Tu_ms / 1000.0;

    // Ziegler-Nichols Tuning Rules
    Kp_base = 0.6 * Ku;
    Ki_base = Kp_base / (Tu_sec / 2.0);
    Kd_base = Kp_base * (Tu_sec / 8.0);
}
```

- **Line 391**: This is the **Describing Function** derivation for a relay.

$$K_u = \frac{4d}{\pi a}$$

  Where d is the relay amplitude (`TUNE_RELAY_AMP`) and a is the measured oscillation amplitude (`avg_a`).

- **Lines 394-396**: These are the standard Ziegler-Nichols closed-loop formulas applied to the derived K_u and T_u.

This automated process replaces hours of manual trial-and-error tuning with a 10-second automated routine.

12 Conclusion

The ESP32 Motor Control System successfully demonstrates the integration of classical control theory with modern IoT architecture.

1. **RPM Control**: The system achieved stable bidirectional speed control across the full **-300 to +300 RPM** range, with the non-linear PWM scaling ensuring consistent performance in both forward and reverse directions.

2. **Position Control**: The implementation of a dedicated PID loop with deadband logic allows for precise angle holding, validated by the `runPositionControl` function.

3. **Simulation**: The `mqtt-simulation.js` module proves that the physical system's state can be accurately mirrored in a virtual environment, opening doors for "Digital Twin" applications where physical RPM drives virtual speed and motor angle drives virtual steering.

4. **Tuning**: The comparative analysis confirmed that while Tyler-Rubinson offers superior stability, Ziegler-Nichols provides the responsiveness required for dynamic motor control applications.

This project serves as a robust foundation for more complex robotics applications, such as differential drive robots or robotic arms.

13 References

1. Astrom, K.J., & Hagglund, T. (1984). "Automatic Tuning of Simple Regulators with Specifications on Phase and Amplitude Margins".

2. Ziegler, J.G., & Nichols, N.B. (1942). "Optimum Settings for Automatic Controllers".

3. Tyreus, B.D., & Luyben, W.L. (1992). "Tuning PI Controllers for Integrator/Dead Time Processes".

4. Espressif Systems. "ESP32 Technical Reference Manual".