ECE3709L Final Project: Wireless PID Motor Control with NRF24L01+, I^2C LCD, Sensor & LabVIEW

By Daniel Romero & Verenize Sotelo

Contents

Objective

Implement a closed-loop PID control system for DC motor control. Additionally, integrate multiple sensors, wireless communication via NRF24L01+, and a LabView GUI for real-time monitoring and control.

PID Analysis

PWM Output Calculation:

$$PWM(output) = PWM(base) + K(p) \cdot e(t) + Ki \cdot \int e(t)dt + K(d) \cdot \frac{de(t)}{dt}$$

Where:

e(t) = Error = Setpoint (RPM) – Measured (RPM)
Kp = 2.0 (proportional gain)

Ki = 0.5 (integral gain)

Kd = 0.1 (derivative gain)

PWM_base = 217 (85% duty cycle baseline)

Code Implementation:

```
float computePID(float setpoint, float input, float dt) {
  float error = setpoint - input;
  float P = Kp * error;  // Immediate response to current error
  integral += error * dt;
  integral = constrain(integral, -100, 100);  // Prevent integral windup
```

```
  float I = Ki * integral;  // Accumulated error over time

  float D = 0;
  if (dt > 0) {
    D = Kd * (error - lastError) / dt;  // Rate of change of error
  }
  lastError = error;

  float output = P + I + D;
  return output;
}
```

The PID controller was tuned using the Ziegler-Nichols method, starting with Ki = 0 and Kd = 0 and increasing the proportional gain until sustained oscillations occurred. The proportional term (P) speeds up response by reacting to current errors, the integral term (I) corrects steady-state error by summing past errors, and the derivative term (D) reduces overshoot by responding to the error's rate of change. This approach provides initial PID settings for stable, responsive control. Figure 1 and Figure 2 show approximate step response before and after tuning for DC motor.

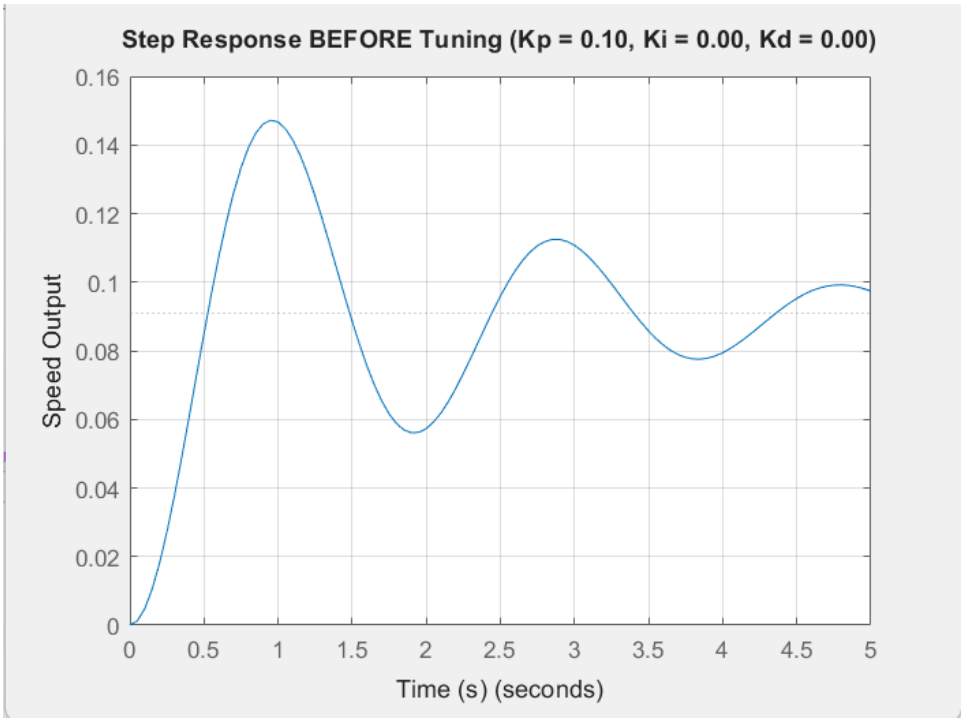


Figure 1: Step Response before tuning

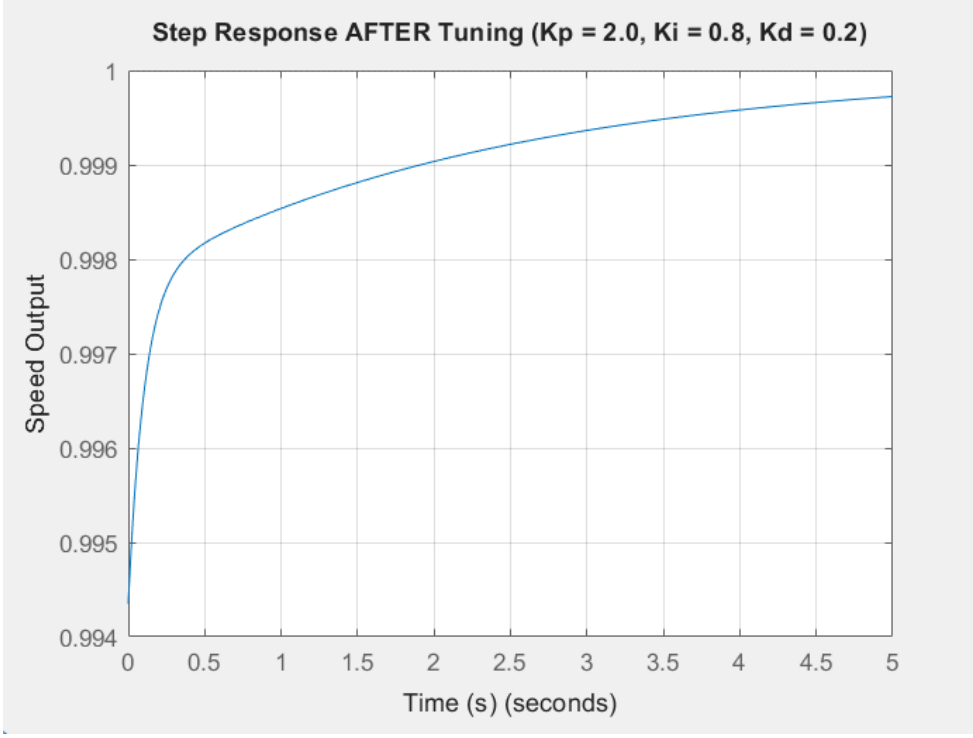


Figure 2: Step Response after tuning

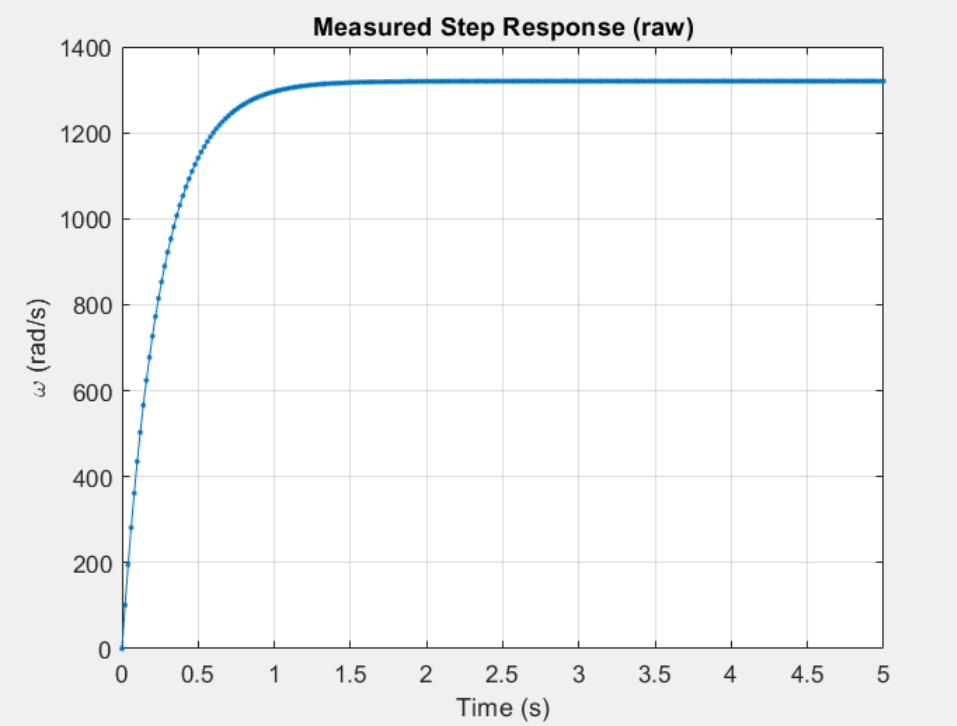


Figure 3: Ideal Step Response

****These graphs are not an accurate representation of our curves and are only meant to show conceptual knowledge as we were not able to get our actual curve data to graph before submission of this report.**

Ziegler-Nichols Ultimate Gain Method

PID gains approach:

Ultimate Gain Test:

- Set Ki = 0, Kd = 0
- Increased Kp incrementally: 0.5 → 1.0 → 1.5 → 2.0 → 2.5
- Observed sustained oscillation at Kp = 2.5
- **Result: Ku ≈ 2.5**

Period Measurement:

- Measured oscillation period from Serial Monitor timestamps
- **Result: Tu ≈ 0.8 seconds**

Initial Gain Calculation (Ziegler-Nichols):

- Kp = 0.6 × 2.5 = 1.5
- Ki = 2 × 1.5 / 0.8 = 3.75
- Kd = 1.5 × 0.8 / 8 = 0.15

Empirical Fine-Tuning:

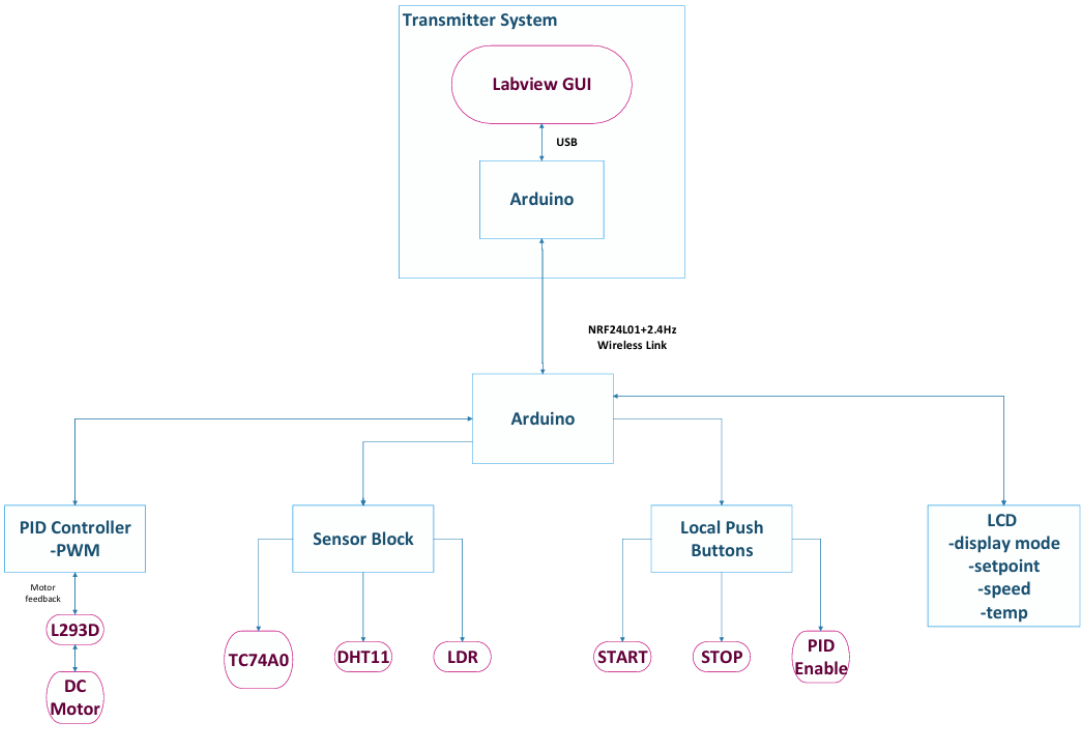
- Z-N gains produced excessive overshoot (~25%)
- Increased Kp to 2.0 for faster initial response
- Reduced Ki to 0.5 to decrease overshoot
- Reduced Kd to 0.1 to minimize noise amplification from encoder

Final Tuned Values:

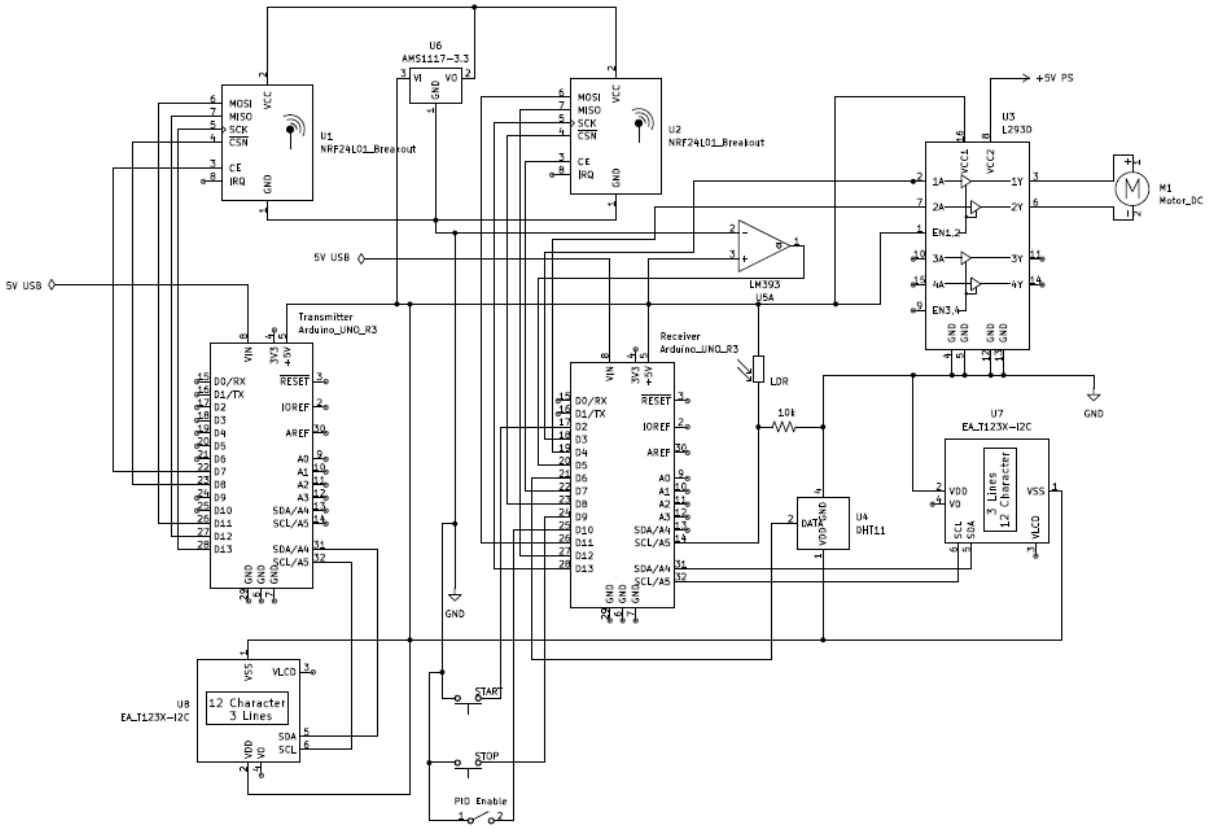
- Kp = 2.0
- Ki = 0.5
- Kd = 0.1

The final gains represent a balance between Z-N theoretical values and empirical adjustments for this specific motor system, prioritizing stability and minimal steady-state error over aggressive response.

Block Diagram



Wiring Diagram



Arduino Code

Transmitter

```
/*
 * nRF24L01 Transmitter with LCD
 * Sends motor commands and displays status on LCD
 * Receives status updates from receiver (including PID mode)
 */

#include <SPI.h>
#include <RF24.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Pin definitions
#define CE_PIN 7
```

```
#define CSN_PIN 8

// Create RF24 object
RF24 radio(CE_PIN, CSN_PIN);

// Create LCD object (address 0x27, 16 columns, 2 rows)
// If 0x27 doesn't work, try 0x3F
LiquidCrystal_I2C lcd(0x27, 16, 2);

// Address through which two modules communicate
const byte txAddress[6] = "00001";  // Transmit to receiver
const byte rxAddress[6] = "00002";  // Receive from receiver

// Current motor status
String motorStatus = "Stop";
String lastMotorStatus = "Stop";
String receiverStatus = "Unknown";
String pidMode = "Unknown";

unsigned long lastStatusRequest = 0;
const unsigned long STATUS_REQUEST_INTERVAL = 500; // Request status every 500ms

void setup() {
 Serial.begin(9600);

 // Initialize LCD
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0, 0);
 lcd.print("TX:Stop");
 lcd.setCursor(0, 1);
 lcd.print("RX:Unknown");

 // Initialize nRF24L01
 if (!radio.begin()) {
  Serial.println("nRF24L01 not responding!");
  while (1);
 }

 radio.openWritingPipe(txAddress);
 radio.openReadingPipe(1, rxAddress);
 radio.setPALevel(RF24_PA_LOW);
 radio.startListening(); // Start in listening mode
```

```
  Serial.println("=== nRF24L01 Transmitter Ready ===");
  Serial.println("Commands: F/f=Forward, R/r=Reverse, S/s=Stop");
}

void updateLCD() {
  lcd.clear();

  // First line: Transmitter command
  lcd.setCursor(0, 0);
  lcd.print("TX:");
  lcd.print(motorStatus);

  // Second line: Receiver status and PID mode
  lcd.setCursor(0, 1);
  lcd.print("RX:");
  lcd.print(receiverStatus);
  lcd.print(" ");
  if (pidMode == "PID_ON") {
    lcd.print("PID");
  } else if (pidMode == "PID_OFF") {
    lcd.print("MAN");
  }
}

void sendCommand(String command) {
  radio.stopListening(); // Switch to transmit mode

  char text[32];
  command.toCharArray(text, sizeof(text));

  bool success = radio.write(&text, sizeof(text));

  if (success) {
    Serial.print("✓ Command sent: ");
    Serial.println(command);
  } else {
    Serial.println("✗ Transmission failed!");
  }

  radio.startListening(); // Switch back to listening mode
  delay(10); // Small delay to ensure mode switch
}
```

```
void loop() {
  unsigned long currentTime = millis();

  // Check for status updates from receiver
  if (radio.available()) {
    char text[32] = "";
    radio.read(&text, sizeof(text));

    String statusUpdate = String(text);
    statusUpdate.trim();

    if (statusUpdate.length() > 0) {
      // Parse message format: "State|PID_MODE"
      int separatorIndex = statusUpdate.indexOf('|');
      if (separatorIndex > 0) {
        receiverStatus = statusUpdate.substring(0, separatorIndex);
        pidMode = statusUpdate.substring(separatorIndex + 1);
      } else {
        // Old format compatibility (just state)
        receiverStatus = statusUpdate;
      }

      updateLCD();
      Serial.print("← Receiver status: ");
      Serial.print(receiverStatus);
      Serial.print(" | PID: ");
      Serial.println(pidMode);
    }
  }

  // Handle serial commands
  if (Serial.available() > 0) {
    char command = Serial.read();

    // Clear any remaining characters in serial buffer
    while(Serial.available() > 0) {
      Serial.read();
    }

    // Process command
    if (command == 'F' || command == 'f') {
      motorStatus = "Forward";
    }
    else if (command == 'R' || command == 'r') {
```

```
    motorStatus = "Reverse";
   }
   else if (command == 'S' || command == 's') {
    motorStatus = "Stop";
   }
   else {
    Serial.println("Invalid command! Use F/R/S");
    return;
   }

   // Send command
   sendCommand(motorStatus);
   lastMotorStatus = motorStatus;
   updateLCD();
  }
}
```

Receiver

```
/*
 * nRF24L01 Receiver with LCD, Sensors, and PID Control
 * Receives motor commands and displays status on LCD
 * Sends status updates back to transmitter
 * Monitors: DHT11, LDR, Encoder
 * PID-controlled motor speed based on LDR input
 */

#include <SPI.h>
#include <RF24.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <DHT.h>

// Pin definitions
#define CE_PIN 7
#define CSN_PIN 8
#define MOTOR_START 5       // Start button on pin 5
#define MOTOR_STOP 9
#define PID_ENABLE_PIN 10

// Motor control pins (L293D with PWM)
#define MOTOR_IN1 3  // L293D pin 2 (PWM capable)
#define MOTOR_IN2 4  // L293D pin 7
#define MOTOR_EN 11  // L293D Enable pin (PWM) - IMPORTANT: Add this pin!
```

```
// Sensor pins
#define DHT_PIN 6
#define LDR_PIN A5
#define ENCODER_PIN 2       // Encoder on pin 2 (interrupt pin)

// DHT sensor type
#define DHT_TYPE DHT11

// Encoder configuration
#define ENCODER_SLOTS 40    // Number of slots in your encoder disk

// Debounce settings
#define DEBOUNCE_DELAY 50  // milliseconds

// PID Configuration
#define BASE_PWM 217        // 85% of 255 (base speed when LDR is uncovered)
#define MAX_PWM 255         // 100% PWM (max speed)
#define MIN_RPM 50          // Minimum target RPM
#define MAX_RPM 200         // Maximum target RPM (adjust based on your motor)

// PID Tuning parameters - adjust these for your system
float Kp = 2.0;   // Proportional gain
float Ki = 0.5;   // Integral gain
float Kd = 0.1;   // Derivative gain

// Create objects
RF24 radio(CE_PIN, CSN_PIN);
LiquidCrystal_I2C lcd(0x27, 16, 2);
DHT dht(DHT_PIN, DHT_TYPE);

// Address through which two modules communicate
const byte rxAddress[6] = "00001";  // Receive commands
const byte txAddress[6] = "00002";  // Send status back

// Motor state tracking
String currentState = "Stop";
String lastSentState = "";
bool lastSentPidState = false;

// Button debouncing variables
int startButtonState = HIGH;
int lastStartButtonState = HIGH;
unsigned long lastStartDebounceTime = 0;
```

```
int stopButtonState = HIGH;
int lastStopButtonState = HIGH;
unsigned long lastStopDebounceTime = 0;

int pidEnableState = HIGH;
int lastPidEnableState = HIGH;
unsigned long lastPidDebounceTime = 0;
bool pidEnabled = false;

// Sensor variables
float temperature = 0;
float humidity = 0;
int ldrValue = 0;
int ldrPercentage = 0;

// Encoder variables
volatile unsigned long encoderPulseCount = 0;
volatile unsigned long lastPulseTime = 0;
float rpm = 0.0;
unsigned long lastRpmCalculation = 0;
unsigned long rpmCalculationInterval = 100; // Calculate RPM every 100ms for better PID response

// PID variables
float targetRPM = 0;
float pidOutput = 0;
float lastError = 0;
float integral = 0;
unsigned long lastPidUpdate = 0;
int currentPWM = 0;

// Timing variables
unsigned long lastSensorRead = 0;
unsigned long lastDisplayUpdate = 0;
const unsigned long SENSOR_READ_INTERVAL = 2000;  // Read DHT every 2 seconds
const unsigned long DISPLAY_UPDATE_INTERVAL = 1000; // Update display every 1 second (slower)

// Display mode - simplified to just show main status
int displayMode = 0;
String lastDisplayLine0 = "";
String lastDisplayLine1 = "";
```

```
// LCD control flag
bool enableLCD = true;  // Set to false to disable LCD if it causes problems

void setup() {
 Serial.begin(9600);

 // Initialize button pins with INPUT_PULLUP
 pinMode(MOTOR_START, INPUT_PULLUP);
 pinMode(MOTOR_STOP, INPUT_PULLUP);
 pinMode(PID_ENABLE_PIN, INPUT_PULLUP);

 // Initialize encoder pin with interrupt
 pinMode(ENCODER_PIN, INPUT_PULLUP);
 attachInterrupt(digitalPinToInterrupt(ENCODER_PIN), encoderISR, FALLING);

 // Initialize motor control pins
 pinMode(MOTOR_IN1, OUTPUT);
 pinMode(MOTOR_IN2, OUTPUT);
 pinMode(MOTOR_EN, OUTPUT);
 digitalWrite(MOTOR_IN1, LOW);
 digitalWrite(MOTOR_IN2, LOW);
 analogWrite(MOTOR_EN, 0);

 // Initialize DHT sensor
 dht.begin();

 // Initialize LCD
 if (enableLCD) {
  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Initializing...");
 }

 delay(1000);

 // Initialize nRF24L01
 if (!radio.begin()) {
  Serial.println("nRF24L01 not responding!");
  if (enableLCD) {
   lcd.setCursor(0, 1);
   lcd.print("RF24 FAIL!");
  }
```

```
  while (1);
 }

 radio.openWritingPipe(txAddress);
 radio.openReadingPipe(1, rxAddress);
 radio.setPALevel(RF24_PA_LOW);
 radio.startListening();

 if (enableLCD) {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Status: Stop");
  lcd.setCursor(0, 1);
  lcd.print("PID: Disabled");
 }

 Serial.println("=== nRF24L01 Receiver Ready ===");
 Serial.println("Pin Configuration:");
 Serial.println("  Encoder: D2 (Interrupt)");
 Serial.println("  Start Button: D5");
 Serial.println("  Stop Button: D9");
 Serial.println("  PID Enable: D10");
 Serial.println("  Motor Enable (PWM): D11");
 Serial.println("  DHT11: D6");
 Serial.println("  LDR: A5");
 Serial.println("PID Tuning: Kp=" + String(Kp) + " Ki=" + String(Ki) + " Kd=" + String(Kd));
 Serial.println("LCD Updates: " + String(enableLCD ? "Enabled" : "Disabled (Serial only)"));
 Serial.println("Waiting for commands...");
}

// Interrupt service routine for encoder
void encoderISR() {
 encoderPulseCount++;
 lastPulseTime = millis();
}

void calculateRPM() {
 static unsigned long lastPulseCount = 0;
 unsigned long currentTime = millis();

 // Calculate time elapsed since last calculation
 unsigned long timeElapsed = currentTime - lastRpmCalculation;

 if (timeElapsed >= rpmCalculationInterval) {
```

```
    // Disable interrupts while reading volatile variable
    noInterrupts();
    unsigned long currentPulseCount = encoderPulseCount;
    interrupts();

    // Calculate pulses in this interval
    unsigned long pulsesThisInterval = currentPulseCount - lastPulseCount;

    // Calculate RPM
    if (timeElapsed > 0) {
      rpm = (float)(pulsesThisInterval * 60000.0) / (float)(ENCODER_SLOTS * timeElapsed);
    } else {
      rpm = 0;
    }

    // If no pulses detected in the last 2 seconds, assume motor is stopped
    if (currentTime - lastPulseTime > 2000) {
      rpm = 0;
    }

    // Update tracking variables
    lastPulseCount = currentPulseCount;
    lastRpmCalculation = currentTime;
  }
}

float computePID(float setpoint, float input, float dt) {
  // Calculate error
  float error = setpoint - input;

  // Proportional term
  float P = Kp * error;

  // Integral term (with anti-windup)
  integral += error * dt;
  // Limit integral to prevent windup
  integral = constrain(integral, -100, 100);
  float I = Ki * integral;

  // Derivative term
  float D = 0;
  if (dt > 0) {
    D = Kd * (error - lastError) / dt;
  }
```

```
  lastError = error;

  // Calculate total output
  float output = P + I + D;

  return output;
}

void updateMotorPWM() {
  // Read LDR every cycle for responsive control
  ldrValue = analogRead(LDR_PIN);

  if (currentState == "Forward" && pidEnabled) {
    unsigned long currentTime = millis();
    float dt = (currentTime - lastPidUpdate) / 1000.0; // Convert to seconds

    if (dt >= 0.05) { // Update every 50ms
      // Map LDR value to target RPM
      // Lower LDR value (more covered) = higher target RPM
      targetRPM = map(ldrValue, 0, 1023, MAX_RPM, MIN_RPM);
      targetRPM = constrain(targetRPM, MIN_RPM, MAX_RPM);

      // Compute PID
      pidOutput = computePID(targetRPM, rpm, dt);

      // Calculate PWM: Base PWM + PID correction
      currentPWM = BASE_PWM + (int)pidOutput;
      currentPWM = constrain(currentPWM, BASE_PWM, MAX_PWM);

      // Apply PWM to motor
      analogWrite(MOTOR_EN, currentPWM);

      lastPidUpdate = currentTime;
    }
  } else if (currentState == "Forward" && !pidEnabled) {
    // Manual control: Map LDR directly to PWM (85% to 100%)
    currentPWM = map(ldrValue, 1023, 0, BASE_PWM, MAX_PWM);
    currentPWM = constrain(currentPWM, BASE_PWM, MAX_PWM);
    analogWrite(MOTOR_EN, currentPWM);
  } else if (currentState == "Reverse") {
    // Simple full speed reverse
    analogWrite(MOTOR_EN, 200);
    currentPWM = 200;
  } else {
```

```
    // Stop
    analogWrite(MOTOR_EN, 0);
    currentPWM = 0;
    integral = 0; // Reset integral when stopped
    lastError = 0;
  }
}

void readSensors() {
  // Read DHT11 (slow operation, do less frequently)
  humidity = dht.readHumidity();
  temperature = dht.readTemperature();

  // Check if DHT readings failed
  if (isnan(humidity) || isnan(temperature)) {
    humidity = 0;
    temperature = 0;
  }

  // Calculate RPM
  calculateRPM();

  // Calculate LDR percentage
  ldrPercentage = map(ldrValue, 0, 1023, 0, 100);
}

void updateDisplaySimple() {
  if (!enableLCD) return;  // Skip if LCD disabled

  // Only show the most important info
  String line0 = currentState.substring(0, 4) + " " + String((currentPWM * 100) / 255) + "% " +
(pidEnabled ? "PID" : "MAN");
  String line1 = "RPM:" + String((int)rpm) + " L:" + String(ldrPercentage) + "%";

  // Pad strings to 16 characters to avoid leftover characters
  while (line0.length() < 16) line0 += " ";
  while (line1.length() < 16) line1 += " ";

  // Only update LCD if content changed
  if (line0 != lastDisplayLine0) {
    lcd.setCursor(0, 0);
    lcd.print(line0);
    lastDisplayLine0 = line0;
  }
```

```
  if (line1 != lastDisplayLine1) {
    lcd.setCursor(0, 1);
    lcd.print(line1);
    lastDisplayLine1 = line1;
  }
}

void sendStatusUpdate() {
  // Send update if state or PID mode has changed
  if (currentState != lastSentState || pidEnabled != lastSentPidState) {
    radio.stopListening();

    // Create status message: "State|PID"
    String statusMsg = currentState + "|" + (pidEnabled ? "PID_ON" : "PID_OFF");
    char text[32];
    statusMsg.toCharArray(text, sizeof(text));

    bool success = radio.write(&text, sizeof(text));

    if (success) {
      Serial.print("→ Status sent: ");
      Serial.println(statusMsg);
      lastSentState = currentState;
      lastSentPidState = pidEnabled;
    }

    radio.startListening();
    delay(5);
  }
}

void setMotorState(String newState, String source) {
  currentState = newState;

  if (currentState == "Forward") {
    digitalWrite(MOTOR_IN1, HIGH);
    digitalWrite(MOTOR_IN2, LOW);
    // PWM will be set by updateMotorPWM()
  }
  else if (currentState == "Reverse") {
    digitalWrite(MOTOR_IN1, LOW);
    digitalWrite(MOTOR_IN2, HIGH);
    analogWrite(MOTOR_EN, 200);
```

```
  }
  else { // Stop
    digitalWrite(MOTOR_IN1, LOW);
    digitalWrite(MOTOR_IN2, LOW);
    analogWrite(MOTOR_EN, 0);
  }

  Serial.print("State changed to: ");
  Serial.print(currentState);
  Serial.print(" (");
  Serial.print(source);
  Serial.println(")");

  sendStatusUpdate();
}

void loop() {
  unsigned long currentTime = millis();

  // CRITICAL: Calculate RPM and update motor EVERY loop iteration
  // These must be fast and non-blocking
  calculateRPM();
  updateMotorPWM();

  // ===== START BUTTON DEBOUNCING =====
  int startReading = digitalRead(MOTOR_START);

  if (startReading != lastStartButtonState) {
    lastStartDebounceTime = currentTime;
  }

  if ((currentTime - lastStartDebounceTime) > DEBOUNCE_DELAY) {
    if (startReading != startButtonState) {
      startButtonState = startReading;

      if (startButtonState == LOW) {
        Serial.println(">>> START BUTTON PRESSED <<<");
        setMotorState("Forward", "Button");
      }
    }
  }

  lastStartButtonState = startReading;
```

```
// ===== STOP BUTTON DEBOUNCING =====
int stopReading = digitalRead(MOTOR_STOP);

if (stopReading != lastStopButtonState) {
  lastStopDebounceTime = currentTime;
}

if ((currentTime - lastStopDebounceTime) > DEBOUNCE_DELAY) {
  if (stopReading != stopButtonState) {
    stopButtonState = stopReading;

    if (stopButtonState == LOW) {
      Serial.println(">>> STOP BUTTON PRESSED <<<");
      setMotorState("Stop", "Button");
    }
  }
}

lastStopButtonState = stopReading;

// ===== PID ENABLE BUTTON DEBOUNCING =====
int pidReading = digitalRead(PID_ENABLE_PIN);

if (pidReading != lastPidEnableState) {
  lastPidDebounceTime = currentTime;
}

if ((currentTime - lastPidDebounceTime) > DEBOUNCE_DELAY) {
  if (pidReading != pidEnableState) {
    pidEnableState = pidReading;

    if (pidEnableState == LOW) {
      pidEnabled = !pidEnabled;
      integral = 0;
      lastError = 0;
      Serial.print(">>> PID BUTTON PRESSED: ");
      Serial.println(pidEnabled ? "ENABLED" : "DISABLED");
      sendStatusUpdate();
    }
  }
}

lastPidEnableState = pidReading;
```

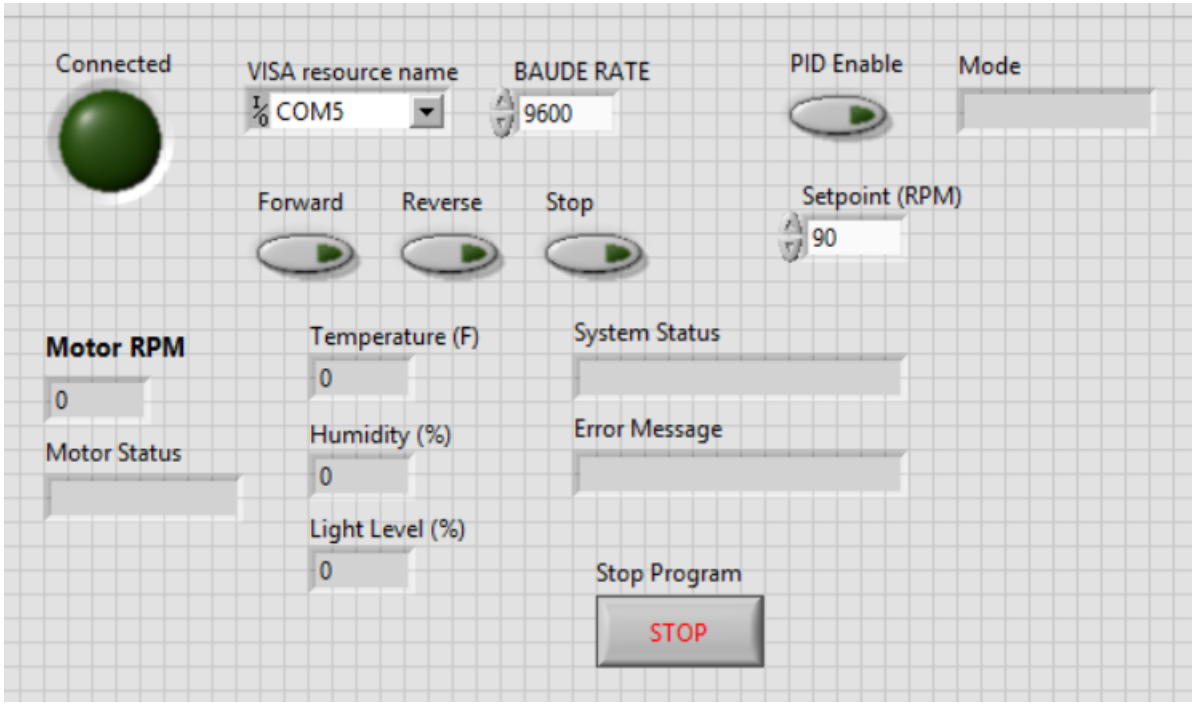
```
// ===== READ SENSORS PERIODICALLY =====
if (currentTime - lastSensorRead >= SENSOR_READ_INTERVAL) {
  readSensors();
  lastSensorRead = currentTime;

  // Print comprehensive status to serial
  Serial.print("LDR:");
  Serial.print(ldrValue);
  Serial.print(" (");
  Serial.print(ldrPercentage);
  Serial.print("%) RPM:");
  Serial.print(rpm, 1);
  Serial.print(" Target:");
  Serial.print(targetRPM, 1);
  Serial.print(" PWM:");
  Serial.print((currentPWM * 100) / 255);
  Serial.print("% Temp:");
  Serial.print(temperature, 1);
  Serial.print("C Hum:");
  Serial.print(humidity, 0);
  Serial.println("%");
}

// ===== UPDATE DISPLAY PERIODICALLY (very slow, do infrequently) =====
if (enableLCD && (currentTime - lastDisplayUpdate >= DISPLAY_UPDATE_INTERVAL)) {
  updateDisplaySimple();
  lastDisplayUpdate = currentTime;
}

// ===== CHECK FOR RF COMMANDS =====
if (radio.available()) {
  char text[32] = "";
  radio.read(&text, sizeof(text));

  String motorStatus = "";
  for (int i = 0; i < 32 && text[i] != '\0'; i++) {
    if (text[i] >= 32 && text[i] <= 126) {
      motorStatus += text[i];
    }
  }
  motorStatus.trim();

  if (motorStatus == "Forward" || motorStatus == "Reverse" || motorStatus == "Stop") {
    setMotorState(motorStatus, "RF");
```

```
  }
 }
}
```

```
//DHT11 to Digital Pin 6
//LDR connected to Analog A5
//Start button - Digital 5
//Stop Button - Digital 9
//PID Enable Button - Digital 10
//LM393 Encoder Pin 2 (Interrupt)
//Motor Enable (PWM) - Digital 11
```

LabVIEW

Front Panel



Block Diagram