Trinity College – Hartford
Engineering Department
ENGR-323-01 – Embedded Systems Design

DC Motor Control using Arduino with PID controller and LCD

Liu Restrepo Sanabria
Fadhil Ahmed

liu.restreposanabria@trincoll.edu,
fadhil.ahmed@trincoll.edu

Taikang Ning, Ph.D.

May, 2025

Contents

1 Problem Statement

This design laboratory centres on the development and implementation of an embedded system for DC motor speed control, incorporating Pulse-Width Modulation (PWM) and a Proportional-Integral-Derivative (PID) control algorithm. PWM is utilised to modulate the duty cycle, thereby adjusting the rotational speed of the motor in response to varying shaft loads. The system is constructed using an Arduino UNO R4 microcontroller, which executes the PID controller and displays the desired and actual speeds in Revolutions Per Minute (RPM.)

The desired target speed is specified programmatically, and the feedback mechanism enables the PID algorithm to dynamically regulate the driving current, ensuring consistent rotational speed despite variations in load.

2 Design Goals

This project aims to implement a closed-loop DC motor control system using an Arduino UNO R4. The system employs Pulse-Width Modulation (PWM) and a discrete PID (Proportional-Integral-Derivative) controller to maintain a target rotational speed, defined directly in the code.

Feedback from an optical encoder allows real-time adjustment of the driving signal of the motor to compensate for load variations. The actual and target RPM values are displayed on a 16×2 LCD, providing clear visual feedback of system performance. The design prioritises stability, responsiveness, and accuracy across various load conditions.

The specific design goals are listed as follows:

- Use PWM to control motor speed through an H-bridge driver.
- Implement a PID controller in Arduino to minimise RPM error.
- Measure actual motor speed using an optical encoder and interrupt-based pulse counting.
- Display target and actual RPM values on a 16x2 LCD.
- Maintain stable speed tracking despite external disturbances or changing loads.

3 Theory and Background

3.1 Pulse-Width Modulation (PWM)

Pulse-Width Modulation (PWM) is a signal modulation technique used to control the power supplied to electrical devices, particularly in embedded motor control. By rapidly

toggling a digital signal between HIGH and LOW states, the average voltage and power delivered to a load are modulated. The key parameter is the *duty cycle* D, defined as:

$$D = \frac{t_{\text{on}}}{T} \times 100\%$$

where t_{on} is the duration the signal is HIGH within one cycle of the total period T. The effective average voltage delivered to the motor is given by

$$V_{\text{avg}} = D \times V_{\text{in}},$$

where V_{in} is the supply voltage. PWM thus enables efficient speed control without significant thermal loss, as the switching elements operate either in full-on or full-off states [1].

3.2 Proportional–Integral–Derivative (PID) Control

A Proportional–Integral–Derivative (PID) controller is a widely adopted feedback control strategy that aims to minimise the error $e(t)$ between a desired setpoint and a measured process variable. The output of the controller $u(t)$ is determined by the expression below

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{d}{dt} e(t). \tag{1}$$

Each term corresponds to a specific control action:

- **Proportional (K_p)**: scales with present error to provide immediate response.
- **Integral (K_i)**: accounts for cumulative past error to eliminate steady-state bias.
- **Derivative (K_d)**: anticipates future error trends based on its rate of change.

A balance between responsiveness and stability can be achieved by tuning the coefficients. In embedded systems like the Arduino UNO R4, this controller is implemented in discrete form using sampled data [2].

3.3 H-Bridge Motor Driver

An H-bridge is an essential component for driving DC motors, allowing speed and direction control. It consists of four switches arranged in an H-pattern as shown in Fig. 1 that permits current to flow in either direction through the motor with a voltage higher than the one provided by the microcontroller output pin.

When combined with PWM control on the high-side or low-side switches, the H-bridge effectively modulates motor speed while preserving directional flexibility [3]. In this project, a unidirectional implementation is sufficient since direction reversal is not

required, and a Quadruple H-Bridge integrated circuit (IC) is employed; more specifically, an SN754410.

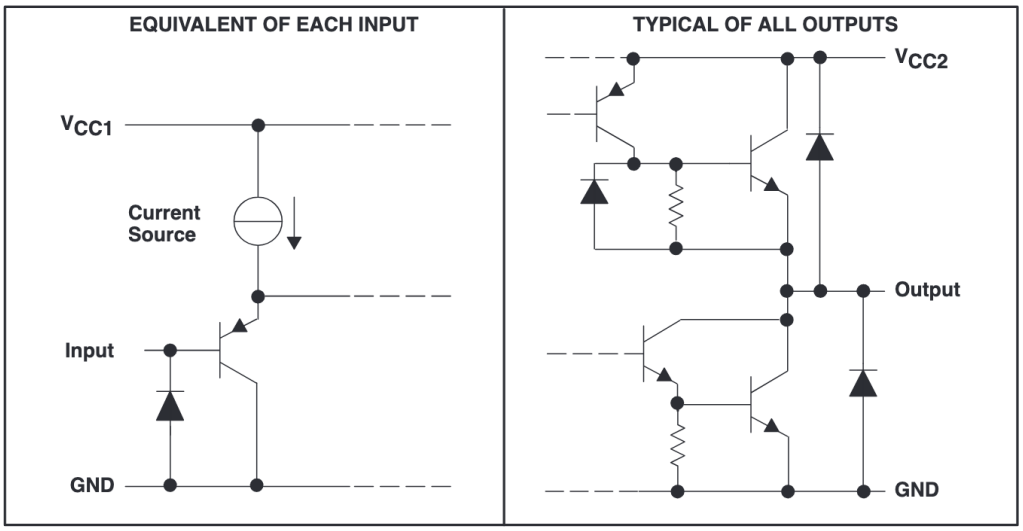

Figure 1: Half-H Driver circuit diagram for input and output for an SN754410

3.4 Rotational Speed Measurement via Encoder

The motor shaft is instrumented with a Hall-effect rotary encoder, producing a square-wave signal where each pulse corresponds to a fixed angular displacement. The actual RPM of the motor is estimated by counting the number of pulses within a defined interval and using the following equation:

$$\text{RPM} = \frac{N_p \times 60}{\text{PPR} \times T}, \tag{2}$$

where N_p is the number of detected pulses, PPR is the pulses per revolution (16 for this system,) and T is the measurement duration in seconds. Interrupts are employed to count pulses with high temporal precision, reducing missed counts due to processor delays.

3.5 LCD Display and Arduino UNO R4

The system provides real-time visual feedback using a 16x2 LCD in 4-bit communication mode. Unlike the traditional 8-bit mode, the 4-bit interface requires fewer I/O pins; only 6 in total: 4 data (DB4 to DB7) + RS and E, thus conserving GPIO resources on the Arduino UNO R4.

This microcontroller, based on the Renesas RA4M1 (Arm Cortex-M4,) offers enhanced performance and flexible timer configurations suited for PWM generation and interrupt handling [4].

4 Design Strategy

4.1 Hardware Design

The hardware design of the embedded DC motor control system integrates a microcontroller (Arduino UNO R4,) an H-bridge motor driver, a DC motor with an optical encoder, and a liquid crystal display (LCD.) The overall circuit architecture enables closed-loop speed regulation through PWM-based actuation and encoder-based feedback.

4.1.1 Microcontroller and Control Logic

At the heart of the system is the Arduino UNO R4, selected for its ease of use, built-in PWM support, and adequate digital I/O resources. Unlike the initially proposed SiliconLabs C8051F120 microcontroller, the UNO R4 simplifies development by providing higher-level abstractions for PWM generation and interrupt handling. The control logic, including PID computation and pulse counting, is fully embedded within the Arduino firmware. The target RPM is hardcoded, eliminating the need for external input interfaces or rotary direction toggles. These are key differences in what Fig. 2 shows, since the figure is provided by the professor before changing the laboratory criteria.

4.1.2 Motor and Encoder Feedback

The DC motor used in this setup includes a built-in Hall-effect rotary encoder. This encoder outputs digital pulses corresponding to shaft rotations. The signal is connected to the external interrupt pin of the Arduino (digital pin 2,) which triggers on rising edges to increment a pulse counter. This allows for real-time estimation of rotational speed, as described in Section 3.4.

4.1.3 PWM and H-Bridge Motor Driver

Motor speed control is achieved by modulating the PWM signal output from digital pin 3 of the Arduino. This signal is fed into the H-bridge driver SN754410, which acts as a power amplifier and routes the control voltage to the motor terminals. As bidirectional rotation is not required for this design, only a single control direction is utilised. The H-bridge is powered via an external variable DC power supply to provide sufficient current for the motor.

4.1.4 LCD Output

A 16×2 character LCD provides visual feedback of the system status. It operates in a 4-bit communication mode to minimise GPIO usage on the Arduino. The display

shows both the hardcoded target RPM and the real-time actual RPM calculated based on encoder pulses.

4.1.5 Integration on CADET Board

The entire circuit was assembled using the CADET development platform, which offers regulated power rails, prototyping space, and convenient routing for peripheral components. A variable DC power supply was connected through the CADET board to ensure stable operation of the motor and H-bridge.

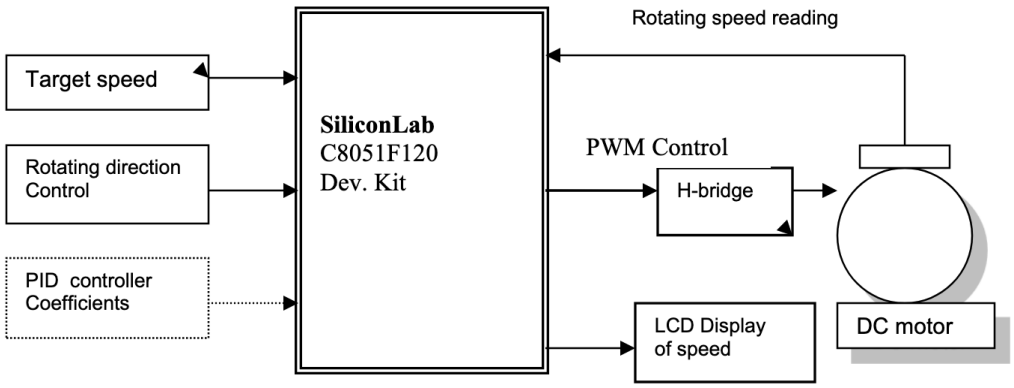

Figure 2: System-level hardware interface showing connections between components. The design presents key differences compared to the figure: no rotating direction control, target speed is hardcoded, and Arduino UNO R4 instead of SiliconLab C8051F120 Dev. Kit.

Table 1 summarises the components used in the implementation of the hardware system.

Table 1: List of Components Used in the DC Motor Control Embedded System

Component	**Quantity**
Arduino UNO R4	1
DC Motor with built-in optical encoder	1
H-Bridge SN754410	1
16×2 LCD Display	1
CADET Board	1
Variable DC Power Supply	1
Miscellaneous (wires, resistors, etc.)	-

4.2 Software Design

The software architecture for the DC motor control system is developed using the Arduino programming environment. It is structured into four functional segments: port and

constant definitions, variable initialisations, setup configuration, and runtime control via the `loop()` function. The program uses timer-based sampling and interrupt-driven pulse counting to measure the rotational speed of the motor, with a PID controller implemented in software to adjust the PWM duty cycle. Key components include encoder-based speed feedback, PWM output, and LCD data display. The full working code can be found in Appendix A.

4.2.1 Pin and Constant Definitions

The code begins by declaring the microcontroller pin assignments for PWM output, pulse counting from the encoder, and LCD interfacing. The encoder is configured to generate interrupts on each rising edge, and the LCD is operated in 4-bit mode using six control and data pins.

Listing 1: Port and Constant Definitions

```
#include <LiquidCrystal.h>

const uint8_t PWM = 3;         // PWM output to H-bridge
const uint8_t PULSES = 2;      // Encoder pulse input (interrupt)
const uint8_t LCD_RS = 8;
const uint8_t LCD_E  = 9;
const uint8_t LCD_D4 = 4;
const uint8_t LCD_D5 = 5;
const uint8_t LCD_D6 = 6;
const uint8_t LCD_D7 = 7;
```

Constants are also defined for the number of encoder pulses per revolution, the target RPM, sampling intervals, and the PID controller gains. The sampling interval is set to 500 ms to ensure stable readings at moderate RPM.

The PID controller computes the control output (PWM duty cycle) based on the difference between the target and actual RPM. Initial controller coefficients were chosen empirically as:

$$K_p = 1.2, \quad K_i = 0.5, \quad K_d = 0.05$$

These values were iteratively tuned using step-response testing:

- Increasing K_p improved rise time but introduced overshoot.
- Adding K_i eliminated steady-state error.
- A small K_d term was introduced to dampen oscillations and improve settling time.

Final tuning provided a fast and stable response with minimal overshoot and negligible steady-state error under moderate load changes. Thus, the final coefficients that ensured

a smooth and stable PID controller were:

$$K_p = 0.08, \quad K_i = 0.022, \quad K_d = 0.02.$$

Listing 2: PID and Timing Constants

```
const uint16_t PULSES_PER_REV = 16;
const uint16_t MAX_RPM = 500;
const uint16_t targetRPM = 1500;
const unsigned long INTERVAL_MS = 500;
const float RPM_CALIBRATION_FACTOR = 1.0;

const float KP = 0.08;
const float KI = 0.022;
const float KD = 0.02;
const float MAX_INTEGRAL = 400.0;
```

4.2.2 Interrupt-Based Pulse Counting

A hardware interrupt is attached to pin D2 to measure the rotational speed of the motor. Each pulse from the encoder increments a counter.

Listing 3: Interrupt Service Routine

```
volatile uint32_t pulseCount = 0;

void EX0_COUNT_ISR() {
  pulseCount++;
}
```

This pulse count is used later to compute RPM using (2).

4.2.3 Setup Function

The `setup()` function performs initialisation of the LCD, configures I/O pins, and enables the interrupt on pin D2 for encoder input. The PWM output is started at 0 duty to prevent initial overshoot.

Listing 4: Setup Configuration

```
void setup() {
  lcd.begin(16, 2);
  lcd.print("RPM DISPLAY");

  pinMode(PWM, OUTPUT);
  analogWrite(PWM, 0);

  pinMode(PULSES, INPUT_PULLUP);
```

```
9   attachInterrupt(digitalPinToInterrupt(PULSES), EX0_COUNT_ISR,
        RISING);
10
11  prevTime = millis();
12 }
```

4.2.4 Control Loop Execution

The `loop()` function is the core of the control system, periodically sampling the encoder signal and adjusting the duty cycle using a PID controller. It operates on a fixed sampling interval of 500 ms and executes the full control logic within that window.

```
1 unsigned long current = millis();
2
3 if (current - prevTime >= INTERVAL_MS) {
4   noInterrupts();
5   uint32_t pulses = pulseCount;
6   pulseCount = 0;
7   interrupts();
```

The above snippet begins by checking whether the defined interval `INTERVAL_MS` has passed since the last RPM computation. If so, the pulse count is read and cleared inside an atomic block to prevent inconsistencies caused by concurrent updates in the interrupt service routine.

The actual RPM is then calculated using (2) with a calibration factor that was used to account for real-world discrepancies that are not captured by the theoretical formula, such as timing inaccuracies, mechanical issues, and encoder imperfections. We kept adjusting the calibration factor and found a reasonably good result with the value of 1. We left the calibration factor code to make it more flexible for other systems that need calibration.

```
1   uint32_t actualRPM = (pulses * 60000 / (PULSES_PER_REV *
        INTERVAL_MS)) * RPM_CALIBRATION_FACTOR;
2   int error = targetRPM - actualRPM;
```

The control error is computed as the difference between the target and actual RPM. A conditional block activates the PID controller only when the error exceeds 2% of the target speed. This deadband reduces unnecessary adjustments and stabilises behaviour during steady-state operation.

```
1   if (abs(error) >= targetRPM * 0.02) {
2     float pTerm = KP * error;
3     integral += error * (INTERVAL_MS / 1000.0);
4     integral = constrain(integral, -MAX_INTEGRAL, MAX_INTEGRAL);
5     float iTerm = KI * integral;
6     float derivative = (error - lastError) / (INTERVAL_MS / 1000.0);
7     float dTerm = KD * derivative;
8     lastError = error;
```

The three components of the PID controller are calculated using (1). The integral term accumulates over time and is bounded to avoid *integral windup*, while the derivative term captures the rate of change of the error. The combined adjustment is constrained to prevent aggressive or unstable shifts in motor behaviour.

```
1      int adjustment = (int)(pTerm + iTerm + dTerm);
2      adjustment = constrain(adjustment, -10, 10);
3      currentDuty += adjustment;
```

Duty cycle output is restricted to a safe range (20% to 80%,) corresponding to the PWM register values of 51 and 204, respectively, as calculated using

$$\text{PWM}_{\text{value}} = \frac{D}{100} \times 255,$$

where D is the duty cycle as a percentage.

```
1    currentDuty = constrain(currentDuty, 51, 204); // 20% to 80% of
        255
2    analogWrite(PWM, currentDuty);
```

Finally, the LCD is updated with the current setpoint and actual RPM, providing immediate visual feedback for validation and tuning.

```
1    lcd.clear();
2    lcd.setCursor(0, 0);
3    lcd.print("Target␣RPM:␣");
4    lcd.print(targetRPM);
5    lcd.setCursor(0, 1);
6    lcd.print("Actual␣RPM:␣");
7    lcd.print(actualRPM);
8
9    prevTime = current;
```

This loop ensures periodic control, accurate measurement, and real-time visualisation of the behaviour of the motor, embodying the full closed-loop system on the Arduino platform.

4.2.5 PWM Control and Duty Cycle

PWM control is handled via `analogWrite()` on pin D3. The duty cycle is scaled between 20% and 80% to ensure the motor is neither underpowered nor overdriven. This range is computed as shown previously, thus leading to

$$D_{\min} = \frac{51}{255} \approx 20\%, \quad D_{\max} = \frac{204}{255} \approx 80\%.$$

Adjustments to `currentDuty` are calculated from the PID correction term and constrained within this safe operating window.

4.2.6 Software Overview

The control system is implemented using the Arduino IDE and written in Arduino Language (which is nothing but C++.) The firmware periodically measures the rotational speed of the motor by counting encoder pulses using hardware interrupts. Every 500 ms, the number of counted pulses is used to calculate the actual RPM.

The error between the user-defined target RPM and the actual RPM is fed into a digital PID controller. The controller calculates the necessary PWM duty cycle, which is applied to the SN754410 H-bridge to adjust the motor speed accordingly.

An I2C-connected 16x2 LCD displays both the `Target RPM` and the `Actual RPM`, updating in near real-time. All time-sensitive operations —encoder counting, display refreshing, and PID updates— are scheduled to run without blocking, ensuring smooth control loop operation.

5 System Implementation

The implementation phase involved carefully wiring the components according to the system design described in the previous sections. The Arduino UNO R4 microcontroller served as the central processing unit, with dedicated connections to the H-bridge motor driver, encoder feedback line, and LCD.

Prior to wiring up altogether, each component was tested, thus employing a *divide-and-conquer* approach to ensure system integrity:

1. The **H-bridge** was powered, and a square wave from a signal generator was fed into the circuit to measure the output and compare it to the input using a digital oscilloscope; hence, the voltage of the power supply was tuned to avoid overheating the circuit. For security reasons, a vent was placed on top of the H-bridge for heat sink purposes.

2. The **LCD** and **Arduino Uno R4** were tested by writing a simple `"Hello world!"` code. This code displayed “Hello” in the first row and “World!” in the second row of the LCD.

3. The **DC motor** was provided a square wave from the signal generator, and by adjusting the duty cycle, the motor was tested. The output of the motor was connected to the oscilloscope to visualise the square wave generated and calculate the pulses per rotation of the motor.

After testing each component, the output pin D3 of the Arduino was connected to the input pin of the SN754410 H-bridge to provide the Pulse-Width Modulation (PWM) signal. The H-bridge output was routed to the DC motor terminals, while its power and

ground lines were connected to a variable DC supply through the CADET board power rails. Proper decoupling and orientation were ensured to avoid voltage mismatch and damage to the H-bridge.

The motor encoder output was connected to pin D2 (INT0) of the Arduino to count pulses via hardware interrupts. This line was configured with an internal pull-up resistor to reduce susceptibility to electrical noise. Encoder pulses were expected to be clean digital square waves, but the group verified the integrity of these signals using a digital oscilloscope.

Prior to running the complete control code, a simple PWM test script was uploaded to the Arduino. This script gradually ramped the duty cycle from 0% to 100% to ensure the motor responded accordingly. An oscilloscope probe was placed at the input of the H-bridge to verify that a clean, frequency-consistent PWM waveform was present. Similarly, the H-bridge output to the motor was monitored to confirm that the switching logic correctly amplified the signal.

Once basic connectivity was validated, the full PID code was uploaded to the Arduino UNO R4. Debugging the software required iterative testing and code inspection. Early issues included incorrect duty cycle limits, misconfigured interrupt handlers, and occasional LCD instability. To isolate these problems, the team employed serial print debugging and used a multimeter to verify PWM output voltages.

A key debugging strategy involved monitoring the LCD to ensure it displayed accurate RPM values. To validate correctness, the RPM readings were compared to theoretical values derived from pulse counts observed with the oscilloscope. If discrepancies arose, calibration factors were adjusted until the computed RPM aligned with measurements. Figure 3 shows the final hardware configuration on the CADET board after full assembly and debugging.

Overall, the system functioned reliably after ensuring clean connections, validating signal paths with the oscilloscope, and fine-tuning the PID control logic through empirical testing. This careful implementation process ensured that both the control accuracy and display responsiveness met the project design goals.

6 Results and Discussion

6.1 System Performance

The PID-controlled system consistently converged to the desired RPM within 1–2 seconds under no-load conditions. During steady-state operation, it maintained the setpoint speed with high accuracy. When a sudden mechanical load was introduced, the controller responded within approximately 500 ms by adjusting the PWM output to restore the target speed. The response featured minimal overshoot (typically less than 5%) and

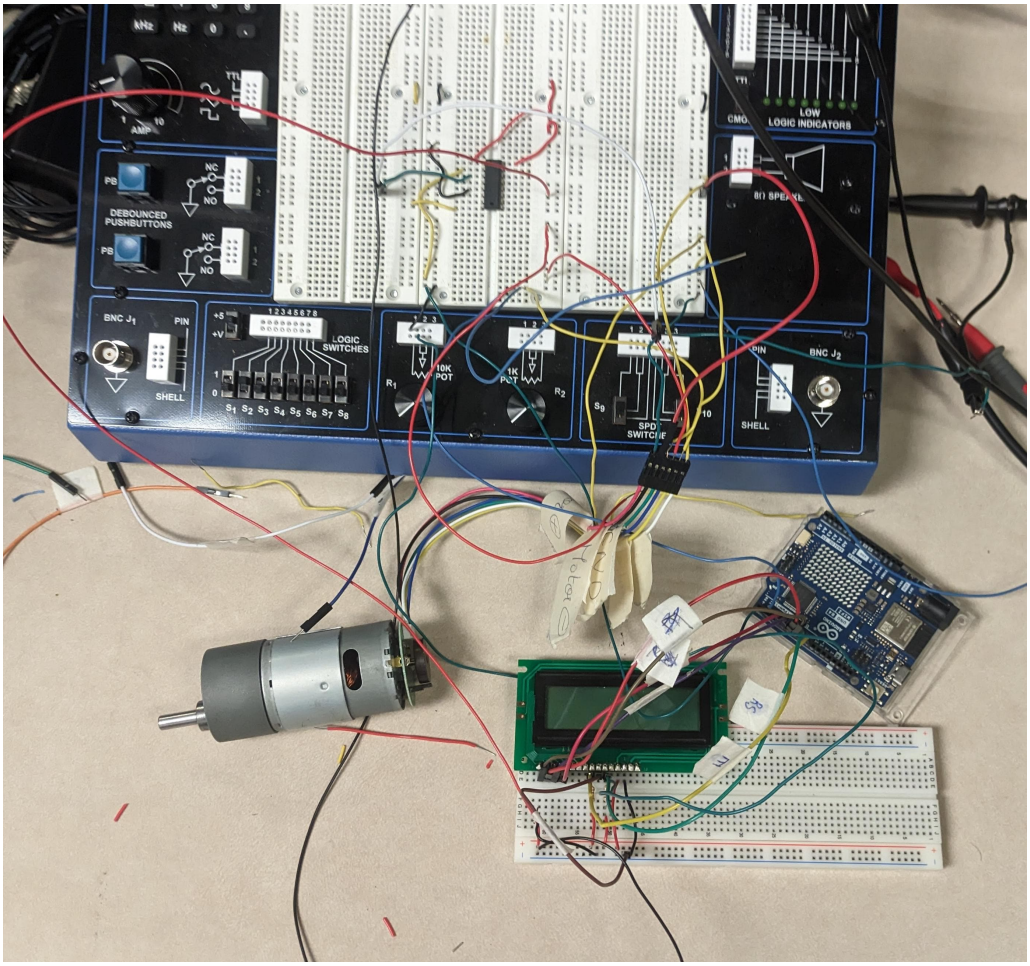

Figure 3: Fully implemented DC motor control system including Arduino UNO R4, LCD, H-bridge driver, and rotary encoder.

quickly settled, validating the effectiveness of the control loop.

6.2 Control Stability

The system exhibited stable closed-loop behavior across varying load conditions. The proportional term provided immediate corrective action, the integral term successfully eliminated steady-state error, and the derivative term dampened the response to avoid oscillations. The result was a critically damped system with robust performance even in the presence of non-linear effects and load disturbances. Integral windup was avoided through bounding, further enhancing reliability.

6.3 Responsiveness and Feedback

Although the target RPM was hardcoded, the system's control response remained fast and consistent throughout testing. The encoder pulse count was sampled every 500 ms, and the calculated RPM was displayed on the LCD with minimal delay or flicker. This

periodic update ensured that both the target and actual RPM values were clearly visible, aiding in performance evaluation and tuning.

However, during extended operation, occasional display glitches were observed due to electrical noise affecting the LCD. To address this, it was sometimes necessary to call `lcd.begin(16, 2)` within the `loop()` function to reinitialise the display. While not ideal, this workaround effectively restored proper operation and ensured consistent visual feedback.

7 Conclusion

This project successfully implemented a closed-loop DC motor control system using an Arduino UNO R4, integrating PWM modulation, encoder-based feedback, and a software-implemented PID controller. The system demonstrated stable and responsive speed regulation under varying load conditions, with real-time feedback displayed on a 16×2 LCD.

The combination of hardware simplicity and effective control logic makes this design a strong foundation for future embedded systems in automation, robotics, and mechatronic applications. The results validate the feasibility of using low-cost microcontrollers for real-time control tasks in educational and prototyping contexts.

8 Future Work

While the current implementation achieved its core objectives, several enhancements can further expand its functionality and robustness:

- **Direction Detection:** Integrate encoder quadrature decoding to determine motor rotation direction, enabling bidirectional control.
- **PID Auto-Tuning:** Apply algorithms such as Ziegler–Nichols or relay feedback to automate PID parameter tuning for varying motors or load profiles.
- **Multi-Motor Coordination:** Extend the system to control and synchronise multiple motors for applications like differential drive robots.
- **User Input Interface:** Add a keypad or rotary encoder to allow users to input and adjust the desired RPM dynamically at runtime.
- **Noise Mitigation:** Improve signal stability and display reliability by applying hardware-level filtering or shielding to reduce electrical interference on the LCD.

References

[1] D. Hwang and J. Lin, "Pulse width modulation (pwm) techniques: A review," *International Journal of Electronics and Electrical Engineering*, vol. 5, no. 3, pp. 145–153, 2017.

[2] K. J. Åström and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2010.

[3] A. S. Sedra and K. C. Smith, *Microelectronic Circuits*. Oxford University Press, 7th ed., 2014.

[4] Arduino Team, "Arduino uno r4 technical specifications." `https://docs.arduino.cc/hardware/uno-r4-wifi`. Accessed April 2025.

A Arduino PID Code

Listing 5: Main Arduino Code with PID

```
#include <LiquidCrystal.h>

// ================== PORT DEFINITIONS ==================
const uint8_t PWM = 3;        // PWM output to H-bridge
const uint8_t PULSES = 2;     // External interrupt pin for pulse
    counting

// LCD pin mapping
const uint8_t LCD_RS = 8;
const uint8_t LCD_E  = 9;
const uint8_t LCD_D4 = 4;
const uint8_t LCD_D5 = 5;
const uint8_t LCD_D6 = 6;
const uint8_t LCD_D7 = 7;

// ================== CONSTANTS ==========================
const uint16_t PULSES_PER_REV = 16;     // Encoder pulses per motor
    revolution
const uint16_t MAX_RPM = 500;           // Maximum expected RPM
const uint16_t targetRPM = 1500;        // Desired RPM
const unsigned long INTERVAL_MS = 500;  // Sampling interval in
    milliseconds
const float RPM_CALIBRATION_FACTOR = 1.0;

const float KP = 0.08;
const float KI = 0.022;
const float KD = 0.02;
const float MAX_INTEGRAL = 400.0;

// ================== VARIABLES ==========================
volatile uint32_t pulseCount = 0;
uint32_t prevTime = 0;
unsigned long lastLCDReset = 0;
const unsigned long LCD_RESET_INTERVAL = 500;

float integral = 0.0;
int lastError = 0;
uint8_t currentDuty = 0;

// Initialize the LCD
LiquidCrystal lcd(LCD_RS, LCD_E, LCD_D4, LCD_D5, LCD_D6, LCD_D7);

// ================== INTERRUPT SERVICE =================
```

```
41 void EX0_COUNT_ISR() {
42   pulseCount++;
43 }
44
45 // ================== SETUP ==============================
46 void setup() {
47   lcd.begin(16, 2);
48   lcd.print("RPM␣DISPLAY");
49
50   pinMode(PWM, OUTPUT);
51   analogWrite(PWM, 0);
52
53   pinMode(PULSES, INPUT_PULLUP);
54   attachInterrupt(digitalPinToInterrupt(PULSES), EX0_COUNT_ISR,
       RISING);
55
56   prevTime = millis();
57 }
58
59 // ================== MAIN LOOP ==========================
60 void loop() {
61   unsigned long current = millis();
62
63   if (current - prevTime >= INTERVAL_MS) {
64     noInterrupts();
65     uint32_t pulses = pulseCount;
66     pulseCount = 0;
67     interrupts();
68
69     // Calculate RPM
70     uint32_t actualRPM = (pulses * 60000 / (PULSES_PER_REV *
         INTERVAL_MS)) * RPM_CALIBRATION_FACTOR;
71     int error = targetRPM - actualRPM;
72
73     // Activate PID only if error exceeds 2\% of target
74     if (abs(error) >= targetRPM * 0.02) {
75       float pTerm = KP * error;
76       integral += error * (INTERVAL_MS / 1000.0);
77       integral = constrain(integral, -MAX_INTEGRAL, MAX_INTEGRAL);
78       float iTerm = KI * integral;
79       float derivative = (error - lastError) / (INTERVAL_MS /
           1000.0);
80       float dTerm = KD * derivative;
81       lastError = error;
82
83       int adjustment = (int)(pTerm + iTerm + dTerm);
84       adjustment = constrain(adjustment, -10, 10);
```

```
85        currentDuty += adjustment;
86      }
87
88      currentDuty = constrain(currentDuty, 51, 204); // Safe PWM duty
            range: 20-80\%
89      analogWrite(PWM, currentDuty);
90
91      // Update LCD
92      lcd.clear();
93      lcd.setCursor(0, 0);
94      lcd.print("Target␣RPM:␣");
95      lcd.print(targetRPM);
96      lcd.setCursor(0, 1);
97      lcd.print("Actual␣RPM:␣");
98      lcd.print(actualRPM);
99
100     prevTime = current;
101   }
102 }
```