Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

RAPORT

Laboratory work No. 5.2

Disciplină: Sisteme Incorporate

Elaborat: Berco Andrei, FAF - 221

Verificat: asist. univ., Martiniuc A.

Chișinău 2025

1. OBJECTIVES

Main purpose of the work: Developing a modular microcontroller application that implements a PID (Proportional-Integral-Derivative) control system for a PC fan, allowing precise RPM control through sensor feedback and PWM output.

The objectives of the work:

- Understanding and implementing PID control principles for real-time physical parameter regulation
- Creating a modular architecture with clear separation between different system components
- Implementing accurate RPM measurement using a tachometer signal from a fan
- Controlling a fan in analog mode through PWM (Pulse Width Modulation)
- Developing a user interface for system monitoring and parameter adjustment
- Creating a data visualization mechanism for analyzing PID controller performance

Problem Definition
This laboratory work implements a PID control system for a PC fan's rotation speed (Variant C – Control of motor speed). The system must:

- Measure rotation speed through a tachometer input connected to an interrupt pin
- Control fan speed through PWM signals, implementing a closed-loop control system
- Allow user adjustment of the target speed (setpoint) through serial commands
- Display current measurements (setpoint, actual RPM, PWM values) in real-time
- Report data in a format compatible with Arduino Serial Plotter for visualization
- Implement additional features like diagnostic tests and alarm indicators

2. DOMAIN ANALYSIS

2.1 Technology stack

Hardware components:

- Arduino UNO microcontroller
- 12V PC fan with 4-pin connector (PWM controllable)
- 10kΩ pull-up resistor for tachometer signal conditioning
- LED for alarm indication (built-in or external with 220Ω resistor)
- Power supply (12V for fan, 5V for Arduino)
- USB cable for programming and serial communication

Software components:

- PlatformIO IDE with Arduino framework
- Custom PID controller implementation
- AVR-specific timer configuration for high-frequency PWM
- Interrupt-based tachometer input processing
- Standard I/O (stdio) formatted output redirected to Serial
- Arduino Serial Plotter for real-time data visualization

2.2 Use cases

HVAC System Fan Control:
This PID-based fan control system can be used in HVAC (Heating, Ventilation, and Air Conditioning) applications where precise airflow control is required. The system can maintain consistent air circulation regardless of changes in system pressure, filter condition, or power supply variations. The PID controller continuously adjusts fan speed to maintain the desired airflow rate, compensating for environmental changes and system degradation over time.

Computer Thermal Management:
Modern computers require precise cooling that balances thermal performance with acoustic comfort. This system can be adapted to provide dynamic fan speed control based on temperature readings, maintaining optimal thermal conditions while minimizing noise. By implementing a PID controller instead of traditional step-based controls, the system can achieve smoother transitions and more stable operating conditions.

Laboratory Equipment:
In laboratory environments where precise air velocity is required (such as in fume hoods, laminar flow cabinets, or test chambers), this system provides controlled and repeatable airflow rates. The diagnostic features and data reporting capabilities make it suitable for scientific applications where consistent documentation of operating conditions is essential.

3. PROJECT DESIGN

3.1 Architecture schema

The architecture of the fan PID control system is organized into interconnected functional blocks that process information from user input to physical output, as shown in Figure 3.1.

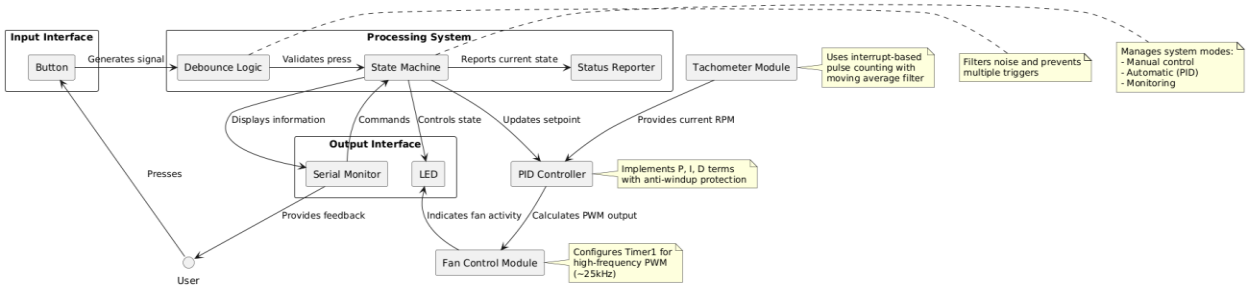


Figure 3.1: System Architecture Schema

The architecture follows a modular design approach where each component has a specific responsibility and interfaces with other components through well-defined functions. The user provides input through the serial interface, which is processed by the command parser. The PID controller continuously calculates the appropriate PWM value based on the error between the target RPM (setpoint) and the actual measured RPM from the tachometer. The fan control module then applies this PWM value to control the fan speed.

The software architecture implements a layered approach, separating application logic from hardware-specific implementations, as shown in Figure 3.2.

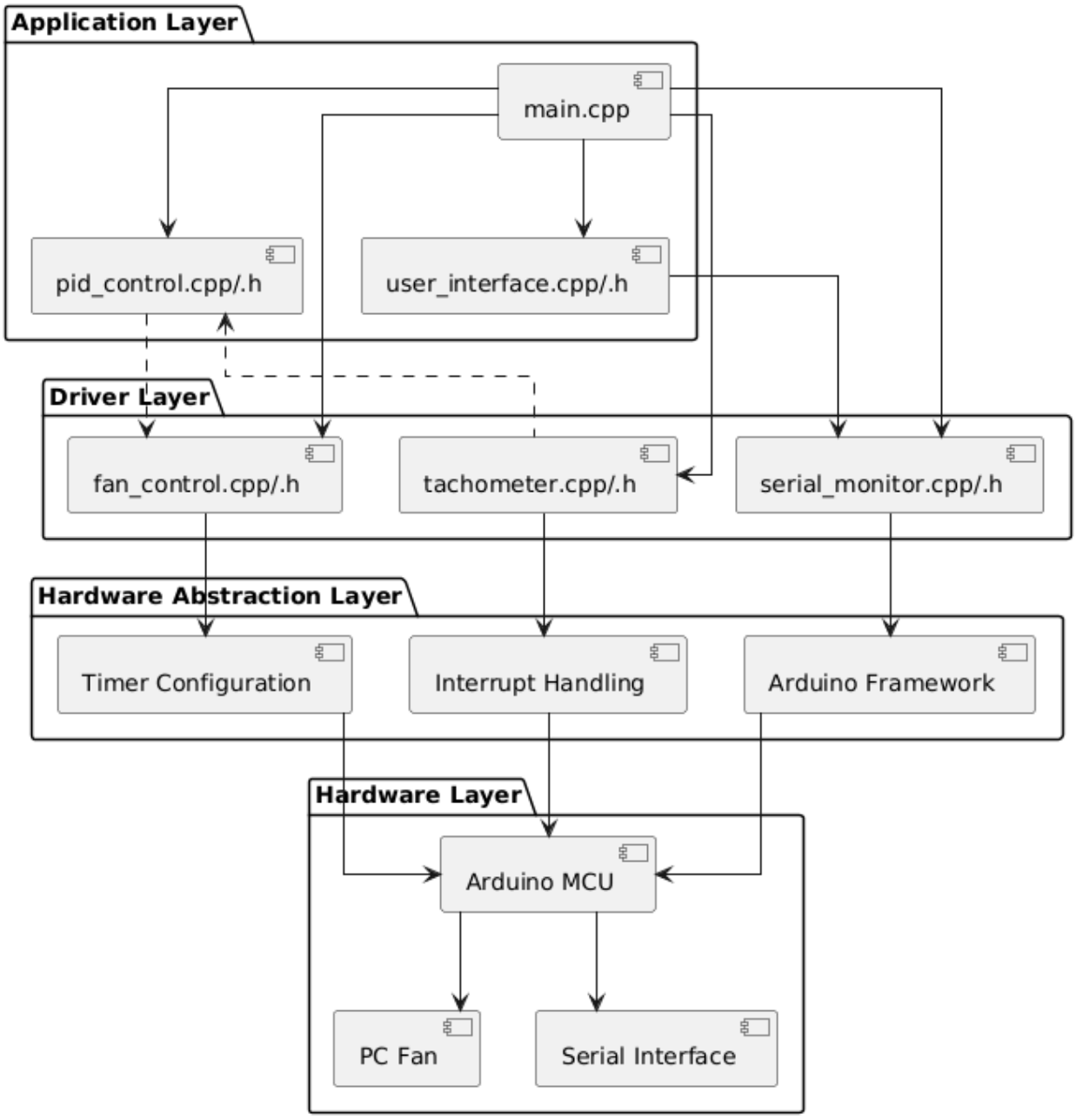


Figure 3.2 Software Architecture Layers

The layered architecture ensures separation of concerns:

1. **Application Layer** contains the main program logic, user interface, and PID algorithm implementation

2. **Driver Layer** provides interfaces to control hardware components (fan, tachometer, serial communication)
3. **Hardware Abstraction Layer** handles the platform-specific implementation details
4. **Hardware Layer** represents the physical components of the system

This structure allows for easy maintenance, testing, and portability to different platforms if needed.

3.2 Flowcharts

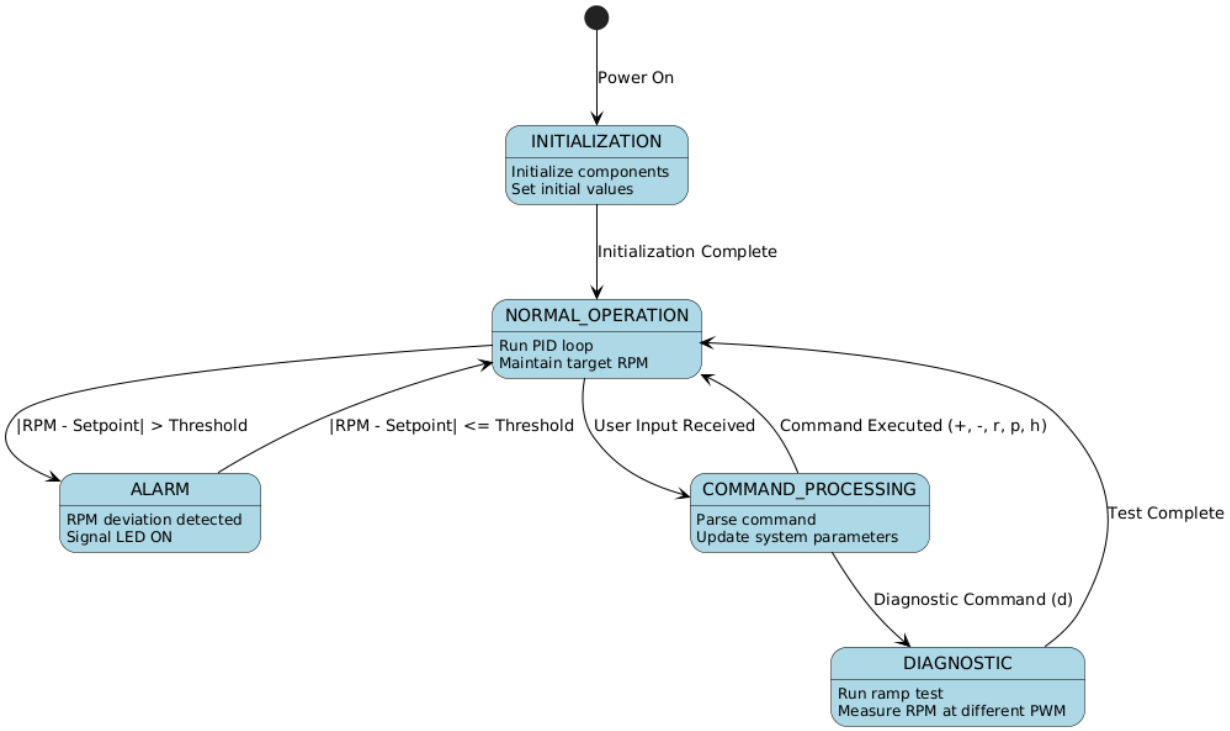


Figure 3.2.1 FSM Diagram

Current State	***Input/Event***	***Next State***	***Output/Action***
START	*Power On*	*INITIALIZATION*	*Initialize serial, fan control, tachometer, PID controller; Set initial values*

INITIALIZATION	*Initialization Complete*	*NORMAL_OPERATION*	*Set fan to initial PWM; Start control loop*
NORMAL_OPERATION	*|RPM - Setpoint| > MAX_RPM_DEVIATION*	*ALARM*	*Turn on alarm LED; Continue PID control*
ALARM	*|RPM - Setpoint| ≤ MAX_RPM_DEVIATION*	*NORMAL_OPERATION*	*Turn off alarm LED; Continue PID control*
NORMAL_OPERATION	*User Input Detected*	*COMMAND_PROCESSING*	*Read and parse command from serial interface*
COMMAND_PROCESSING	*'+' Command*	*NORMAL_OPERATION*	*Increase setpoint by RPM_STEP; Display new setpoint*
COMMAND_PROCESSING	*'-' Command*	*NORMAL_OPERATION*	*Decrease setpoint by RPM_STEP; Display new setpoint*
COMMAND_PROCESSING	*'rXXXX' Command*	*NORMAL_OPERATION*	*Set setpoint to XXXX RPM; Display new setpoint*
COMMAND_PROCESSING	*'p' Command*	*NORMAL_OPERATION*	*Display current parameters; Continue PID control*
COMMAND_PROCESSING	*'h' Command*	*NORMAL_OPERATION*	*Display help message; Continue PID control*

COMMAND_PROCESSING	*'d' Command*	*DIAGNOSTIC*	*Save current PWM value; Display start message*
DIAGNOSTIC	*PWM Ramp Up Complete*	*DIAGNOSTIC*	*Hold at max PWM; Begin ramp down sequence*
DIAGNOSTIC	*PWM Ramp Down Complete*	*NORMAL_OPERATION*	*Restore original PWM; Resume normal operation*
ANY STATE	*Report Timer Elapsed*	*Same State*	*Send data to Serial Plotter; Reset timer*

The main program flow is illustrated in Figure 3.3, showing the initialization process and the main control loop.

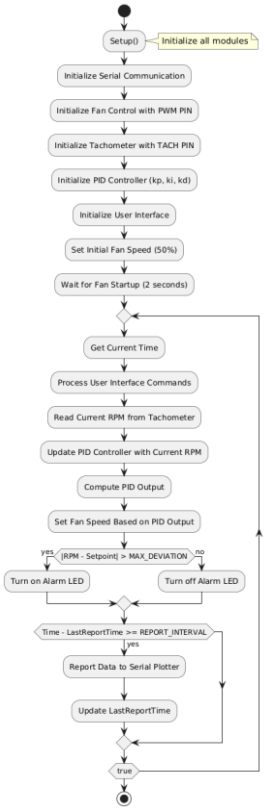


Figure 3.3: Main Program Flowchart

The main program initializes all modules and enters an infinite loop that continuously:

1. Processes user commands from the serial interface
2. Reads the current RPM from the tachometer
3. Updates the PID controller with the current measurement
4. Calculates the appropriate PWM value using the PID algorithm
5. Applies the PWM value to control the fan speed
6. Monitors for alarm conditions (large deviation from setpoint)
7. Reports data periodically for visualization

The PID controller implementation is shown in Figure 3.4, detailing the calculation of the proportional, integral, and derivative terms.

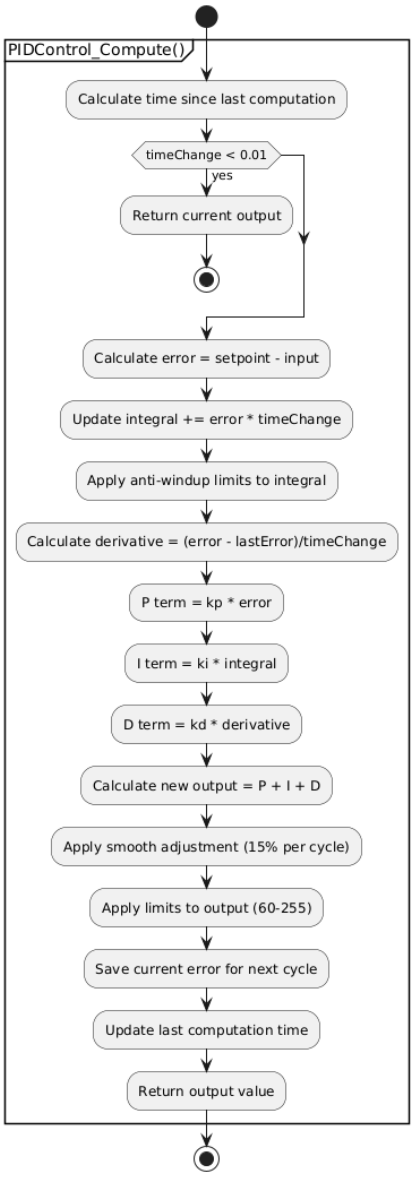


Figure 3.4: PID Controller Flowchart

The PID controller calculates an output value based on:

- **Proportional term**: Responds to the current error (difference between setpoint and measured value)
- **Integral term**: Accumulates past errors to eliminate steady-state error
- **Derivative term**: Responds to the rate of change of error to reduce overshoot

The controller includes anti-windup protection to prevent integral term from growing too large, and applies output limits to ensure the PWM value stays within valid range.

The fan control module, shown in Figure 3.5, handles PWM signal generation for the fan.

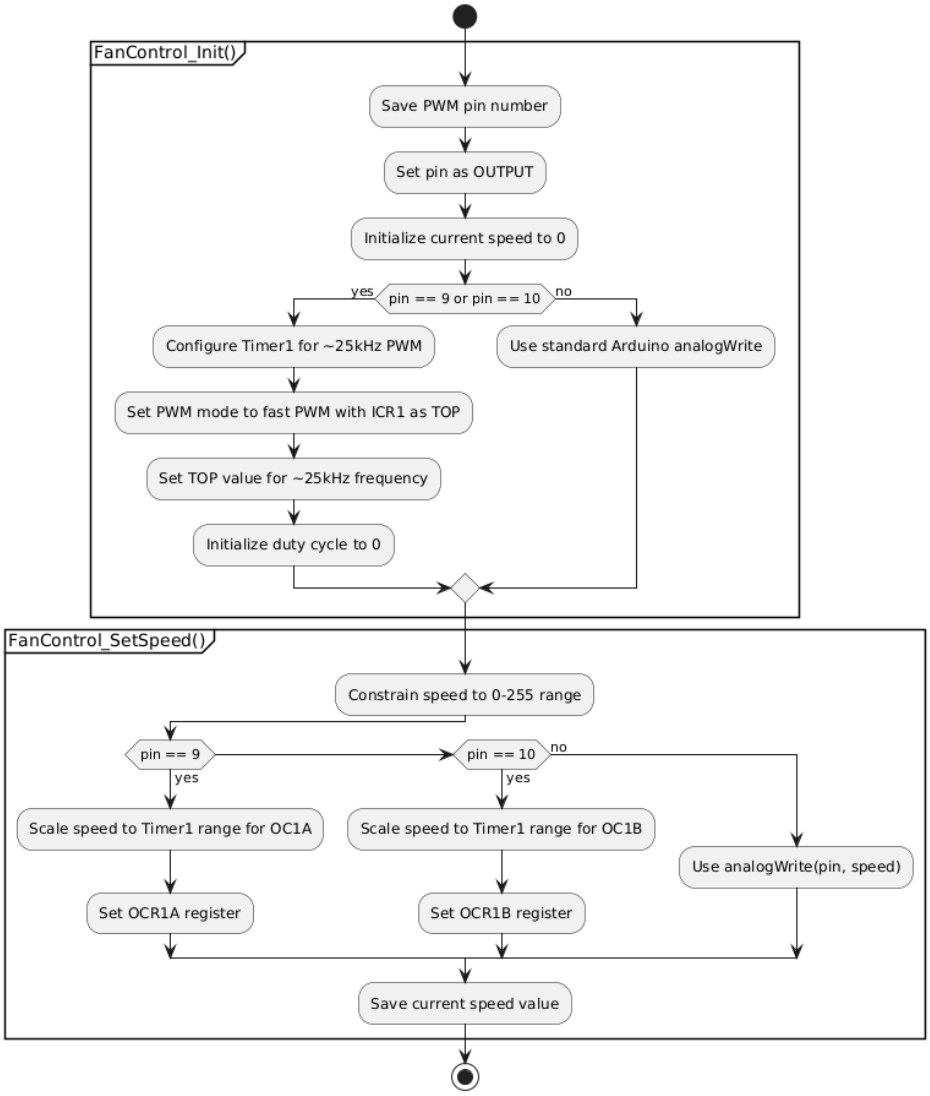


Figure 3.5: Fan Control Module Flowchart

The fan control module configures Timer1 for high-frequency PWM (~25kHz) to avoid audible noise from the fan. It provides a simple interface to set the fan speed using a standard 0-255 value range, which is then mapped to the appropriate timer register values.

The tachometer module, shown in Figure 3.6, handles the measurement of fan RPM through pulse counting.

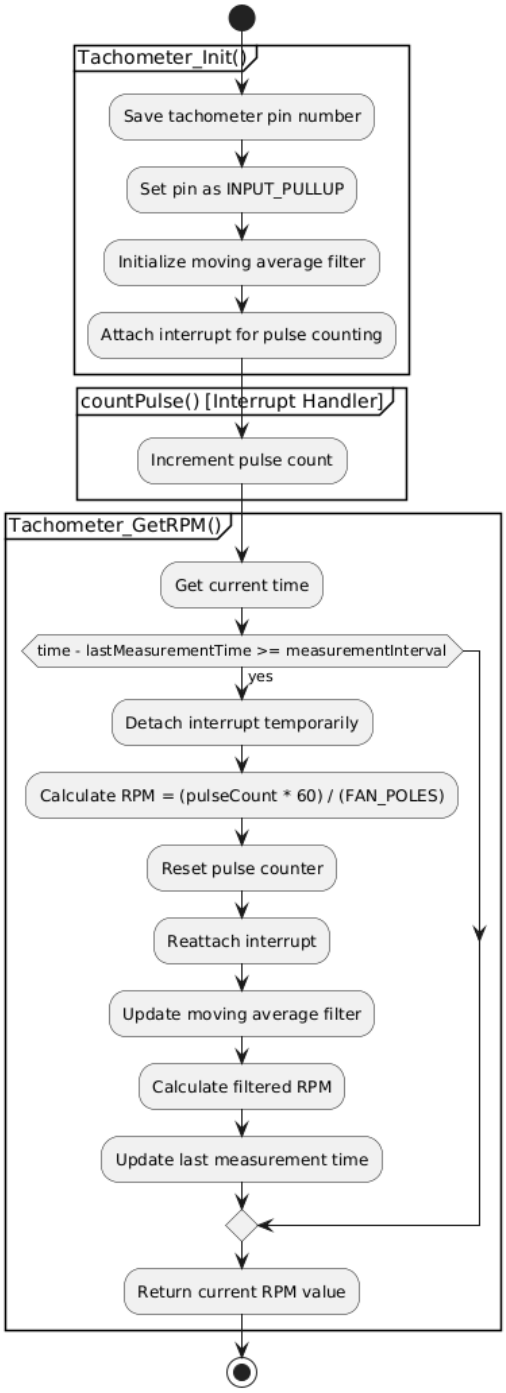


Figure 3.6: Tachometer Module Flowchart

The tachometer module uses an interrupt to count pulses from the fan's tachometer output. It calculates RPM based on the number of pulses received over a fixed time interval, considering the number of poles in the fan motor. A moving average filter is applied to reduce noise in the measurement.

The user interface module, shown in Figure 3.7, handles command processing from the serial interface.

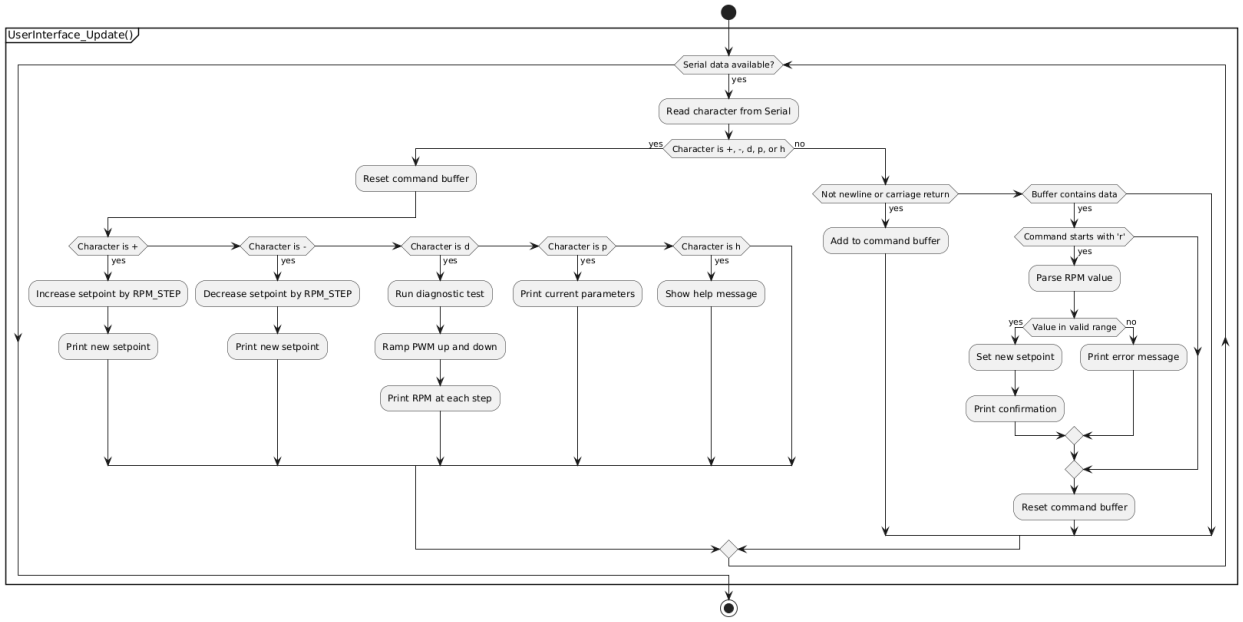


Figure 3.7: User Interface Module Flowchart

The user interface module provides several commands for controlling the system:

- `+` and `-`: Adjust the setpoint by fixed increments
- `rXXXX`: Set the setpoint to a specific RPM value
- `d`: Run a diagnostic test that ramps the PWM up and down
- `p`: Print current system parameters
- `h`: Display help information

3.3 Schematic view of the circuit

The electrical schematic of the fan control system is shown in Figure 3.8.

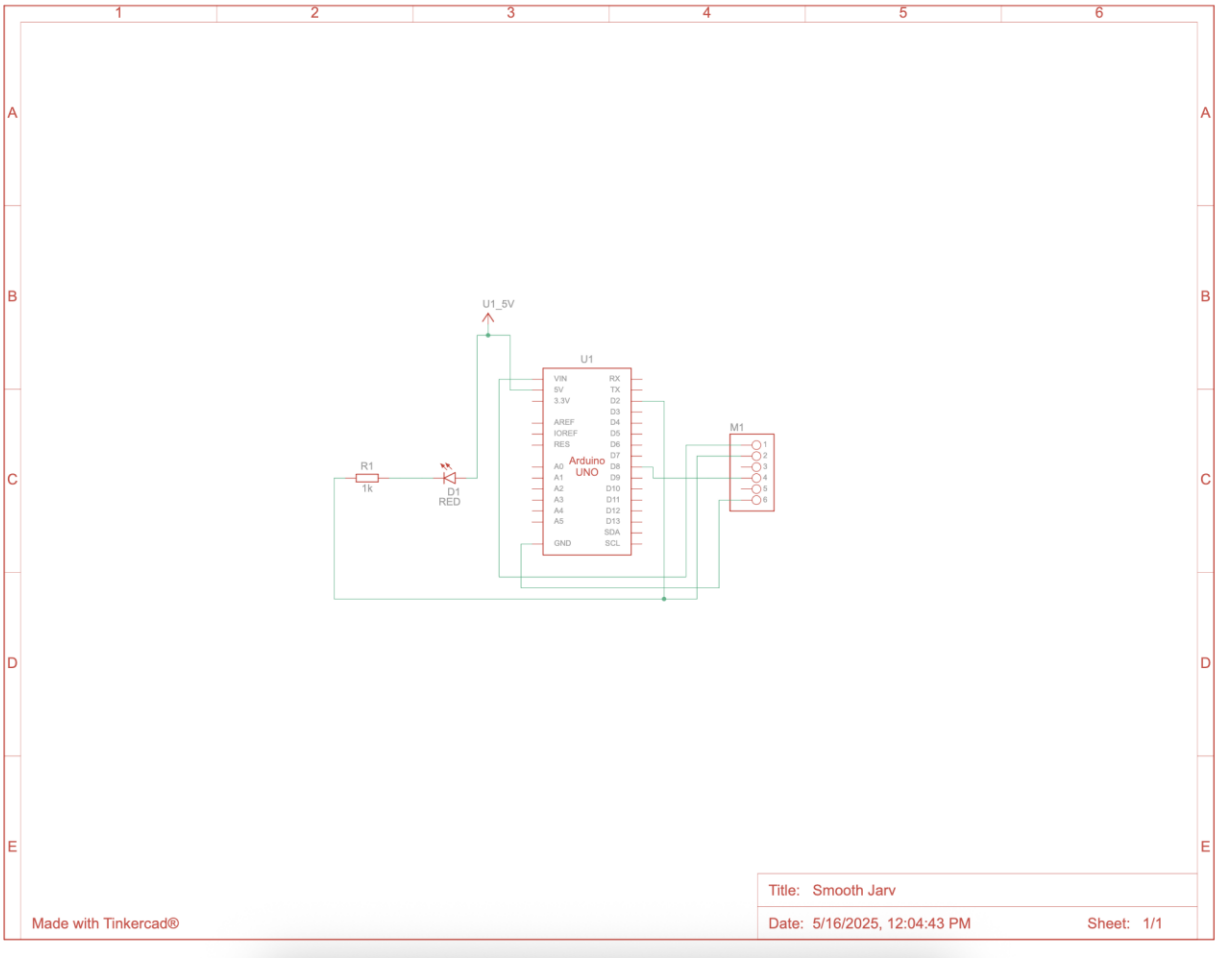
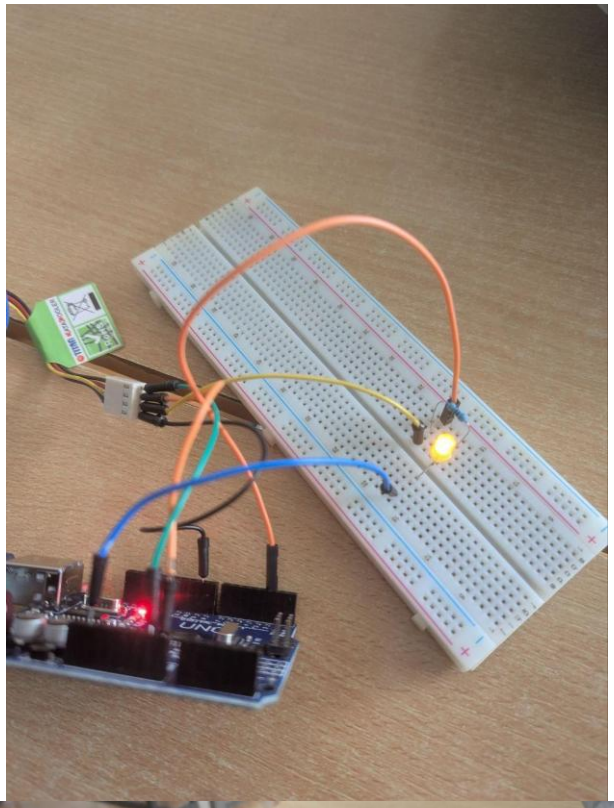
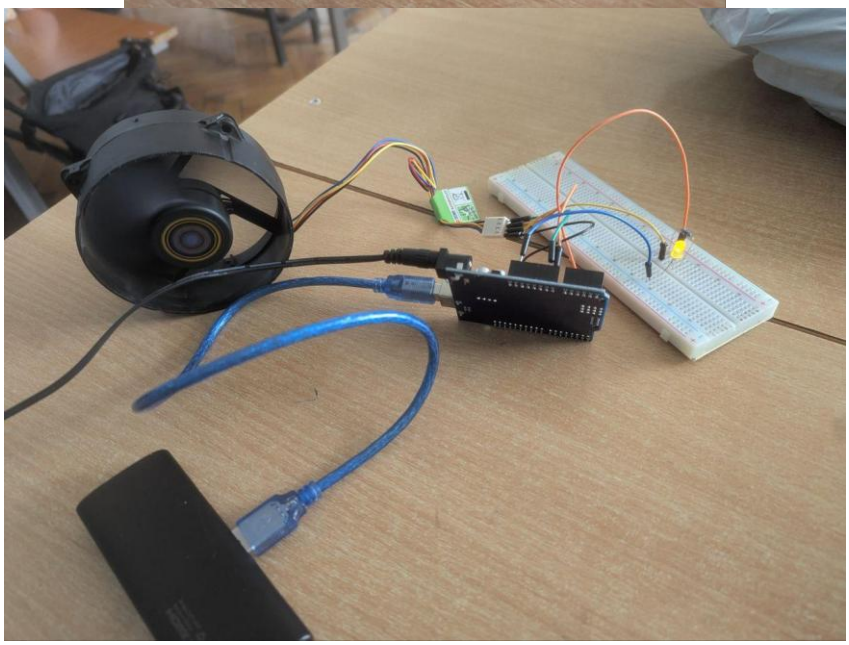


Figure 3.8: Circuit Schematic





The circuit connects the PC fan to the Arduino as follows:

- **Black wire (GND)**: Connected to Arduino GND and power supply GND
- **Red wire (+12V)**: Connected to external 12V power supply
- **Yellow wire (Tachometer)**: Connected to Arduino digital pin 2 (interrupt capable) through a 10kΩ pull-up resistor
- **Blue wire (PWM)**: Connected to Arduino digital pin 9 (PWM capable)

An optional LED is connected to pin 13 to indicate alarm conditions when the RPM deviates significantly from the setpoint.

3.4 Structure of the project

The project follows a modular structure with separate files for each major component:

```
laboratory_5_2/
├── include/
│   ├── fan_control.h       # Fan PWM control interface
│   ├── pid_control.h       # PID controller interface
│   ├── serial_monitor.h    # Serial communication interface with stdio
│   ├── tachometer.h        # RPM measurement interface
│   └── user_interface.h    # Command processing interface
├── platformio.ini          # PlatformIO configuration
├── README.md               # Project documentation
└── src/
    ├── fan_control.cpp     # Fan PWM control implementation
    ├── main.cpp            # Main program flow
    ├── pid_control.cpp     # PID controller implementation
    ├── serial_monitor.cpp  # Serial communication implementation
    ├── tachometer.cpp      # RPM measurement implementation
    └── user_interface.cpp  # Command processing implementation
```

This structure ensures:

- Clear separation of concerns
- Reusability of individual components
- Maintainability through well-defined interfaces
- Testability of each module independently

4. RESULTS

4.1 System Operation and Performance

When the system is started, it initializes all components and begins the PID control loop. Figure 4.1 shows a sample output from the Serial Monitor during startup.

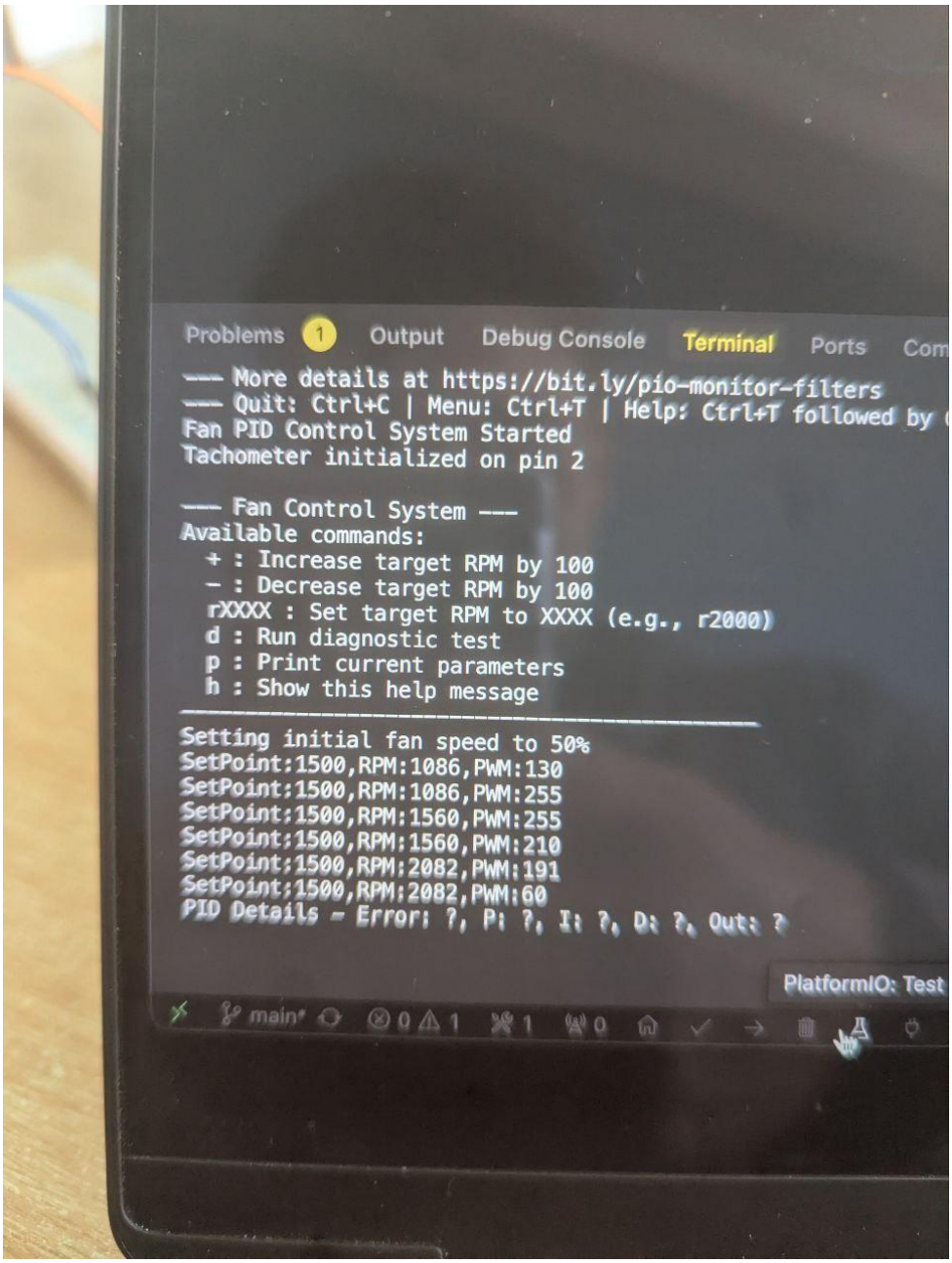


Figure 4.1: System Initialization Output

The system responds to setpoint changes with appropriate adjustments to the PWM output, which in turn affects the fan speed. Figure 4.2 shows the output from Arduino Serial Plotter during a setpoint change from 1500 RPM to 2000 RPM.

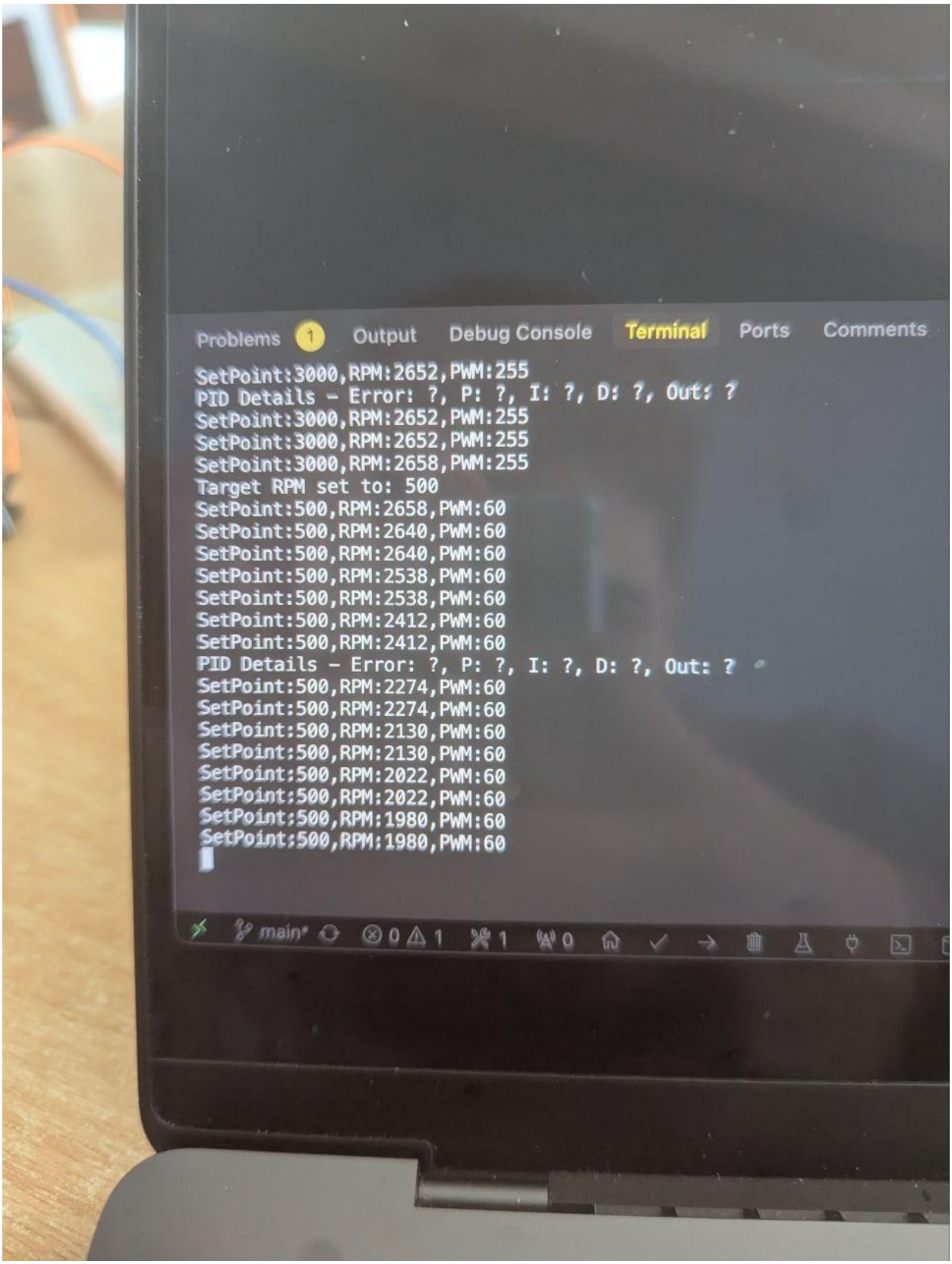


Figure 4.2: PID Response to Setpoint Change

The plot shows three lines:

- **Setpoint** (blue): The target RPM value
- **RPM** (red): The measured RPM from the tachometer
- **PWM** (green): The calculated PWM value (0-255)

When the setpoint is changed, the PID controller increases the PWM output to accelerate the fan. As the measured RPM approaches the setpoint, the controller adjusts the PWM to maintain stable operation. The system demonstrates good stability with minimal overshoot and quick response to setpoint changes.

4.3 System Response to Disturbances

To test the system's ability to handle disturbances, an external force was applied to the fan (simulating increased air resistance). Figure 4.4 shows the system's response.

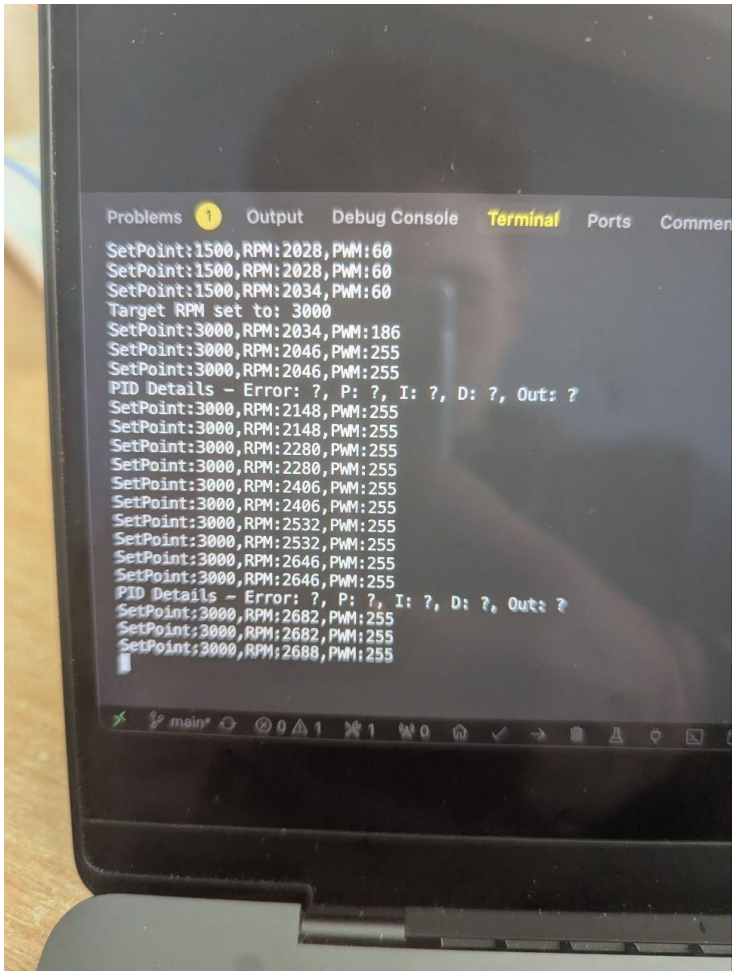


Figure 4.4: System Response to External Disturbance

When the fan is physically obstructed, the RPM drops sharply. The PID controller responds by increasing the PWM output to compensate. Once the obstruction is removed, the controller reduces

the PWM to prevent overshooting the setpoint. This demonstrates the controller's ability to reject disturbances and maintain the desired setpoint under varying conditions.

5. CONCLUSIONS

This laboratory work successfully implemented a PID control system for a PC fan using Arduino. The system demonstrates effective closed-loop control, maintaining a stable RPM despite external disturbances and responding appropriately to setpoint changes.

Key achievements:

1. **Modular Design**: The project was structured with clear separation between components, making it extensible and maintainable.
2. **PID Implementation**: A complete PID controller was implemented with anti-windup protection and tunable parameters.
3. **High-Frequency PWM**: Timer1 was configured for ~25kHz PWM to eliminate audible noise from the fan.
4. **Accurate RPM Measurement**: The tachometer module successfully counts pulses and filters measurements to provide stable RPM readings.
5. **User Interface**: A serial command interface was implemented for interactive control and parameter adjustment.
6. **Visualization**: Data is formatted for Arduino Serial Plotter, allowing real-time visualization of system performance.

Challenges encountered:

1. **Tachometer Signal Conditioning**: The raw tachometer signal required pull-up resistors and filtering to provide stable measurements.
2. **PID Tuning**: Finding optimal PID parameters required experimentation and iterative adjustment.
3. **Timer Configuration**: Configuring Timer1 for high-frequency PWM required direct register manipulation rather than standard Arduino functions.

Learning outcomes:

1. Understanding of PID control theory and practical implementation
2. Experience with interrupt-based sensing for real-time measurements
3. Knowledge of AVR timer configuration for specialized PWM requirements
4. Skills in developing modular embedded software with clear interfaces
5. Techniques for real-time data visualization and analysis

This laboratory work demonstrates the power of closed-loop control systems in maintaining stable operation despite variable conditions, a fundamental concept in many engineering applications.

6. BIBLIOGRAPHY

[1] Arduino - PID Library. URL: https://playground.arduino.cc/Code/PIDLibrary/

[2] ATmega328P Datasheet. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[3] Arduino - PWM. URL: https://www.arduino.cc/en/Tutorial/Foundations/PWM

[4] PlatformIO Documentation. URL: https://docs.platformio.org/en/latest/

[5] Arduino - attachInterrupt(). URL: https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

[6] Åström, K.J. and Hägglund, T., "PID Controllers: Theory, Design, and Tuning", Instrument Society of America, 1995.

[7] Arduino Serial Plotter. URL: https://docs.arduino.cc/software/ide-v1/tutorials/arduino-ide-v1-serial-plotter

7. APPENDIX

fan_control.h

```
#ifndef FAN_CONTROL_H

#define FAN_CONTROL_H


// Initialize fan control

void FanControl_Init(int pwmPin);


// Set fan speed (PWM value 0-255)

void FanControl_SetSpeed(int speed);


// Get current fan speed setting

int FanControl_GetSpeed();


#endif
```

fan_control.cpp

```
#include "fan_control.h"
```

```
#include <Arduino.h>

static int fanPwmPin = 9;  // Default PWM pin
static int currentSpeed = 0;

void FanControl_Init(int pwmPin) {
  fanPwmPin = pwmPin;
  pinMode(fanPwmPin, OUTPUT);
  currentSpeed = 0;

  // Set PWM frequency to ~25kHz (for Timer1, pins 9 and 10)
  // For Arduino Uno/Nano, Timer1 controls pins 9 and 10
  if (fanPwmPin == 9 || fanPwmPin == 10) {
    // Set Timer1 to fast PWM mode (mode 14)
    // WGM13:0 = 1110 for fast PWM mode with ICR1 as TOP
    TCCR1A = _BV(COM1A1) | _BV(COM1B1) | _BV(WGM11);
    TCCR1B = _BV(WGM13) | _BV(WGM12) | _BV(CS10);  // No prescaler

    // Set TOP value for Timer1 to get ~25kHz
    // 16MHz / 25kHz = 640
    ICR1 = 640;

    // Initialize with 0 duty cycle directly
    if (fanPwmPin == 9) {
      OCR1A = 0;
    } else {
      OCR1B = 0;
```

```
    }
  } else {
    // For pins other than 9/10, use standard analogWrite
    analogWrite(fanPwmPin, currentSpeed);
  }
}

void FanControl_SetSpeed(int speed) {
  // Constrain speed to valid PWM range
  speed = constrain(speed, 0, 255);

  // Set PWM value
  if (fanPwmPin == 9) {
    // For pin 9 (OC1A), scale speed from 0-255 to 0-ICR1
    OCR1A = map(speed, 0, 255, 0, ICR1);
  } else if (fanPwmPin == 10) {
    // For pin 10 (OC1B), scale speed from 0-255 to 0-ICR1
    OCR1B = map(speed, 0, 255, 0, ICR1);
  } else {
    // For other pins, use standard analogWrite
    analogWrite(fanPwmPin, speed);
  }

  currentSpeed = speed;
}

int FanControl_GetSpeed() {
```

```
  return currentSpeed;
}
```

pid_control.h

```
#ifndef PID_CONTROL_H
#define PID_CONTROL_H

// Initialize PID controller with tuning parameters
void PIDControl_Init(double kp, double ki, double kd);

// Set the target value (setpoint)
void PIDControl_SetSetpoint(double setpoint);

// Get the current setpoint
double PIDControl_GetSetpoint();

// Set the current input value (measured value)
void PIDControl_SetInput(double input);

// Compute the PID output based on current input and setpoint
double PIDControl_Compute();

#endif
```

pid_control.cpp

```
#include "pid_control.h"
```

```
#include <Arduino.h>

// PID variables
static double kp = 0.5;      // Proportional gain
static double ki = 0.2;      // Integral gain
static double kd = 0.05;     // Derivative gain
static double input = 0;     // Current input value (measured RPM)
static double setpoint = 1500; // Target value (desired RPM)
static double lastError = 0; // Previous error for derivative
static double integral = 0;  // Accumulated error for integral
static double output = 128;  // PID output - start at 50% duty cycle
static unsigned long lastTime = 0; // Last computation time

void PIDControl_Init(double p, double i, double d) {
  kp = p;
  ki = i;
  kd = d;
  input = 0;
  lastError = 0;
  integral = 0;
  output = 128; // Start at midpoint
  lastTime = millis();
}

void PIDControl_SetSetpoint(double sp) {
  setpoint = sp;
  // Reset integral term when setpoint changes to avoid overshooting
```

```
  integral = 0;
}

double PIDControl_GetSetpoint() {
  return setpoint;
}

void PIDControl_SetInput(double in) {
  input = in;
}

double PIDControl_Compute() {
  // Calculate time since last computation
  unsigned long now = millis();
  double timeChange = (double)(now - lastTime) / 1000.0; // Convert to
seconds

  // Don't compute if no time has passed
  if (timeChange < 0.01) {
    return output;
  }

  // Calculate error
  double error = setpoint - input;

  // Calculate integral with anti-windup
  integral += error * timeChange;
```

```
  // Apply stricter limits to integral to prevent windup
  if (integral > 50) integral = 50;
  if (integral < -50) integral = -50;

  // Calculate derivative
  double derivative = 0;
  if (timeChange > 0) {
    derivative = (error - lastError) / timeChange;
  }

  // Calculate PID components
  double pTerm = kp * error;
  double iTerm = ki * integral;
  double dTerm = kd * derivative;

  // Calculate new output
  double newOutput = pTerm + iTerm + dTerm;

  // With simulated RPM, we can map this directly to PWM
  // Map from RPM error to PWM scale
  double pwmAdjustment = newOutput * 255.0 / 3000.0; // Scale based on max
RPM

  // Apply smooth adjustment - 15% of change per cycle
  output += pwmAdjustment * 0.15;

  // Apply limits to output
  if (output > 255) output = 255;
```

```
  if (output < 60) output = 60;  // Minimum threshold to ensure fan runs

  // Occasionally print PID details for debugging
  static unsigned long lastDebugTime = 0;
  if (now - lastDebugTime > 5000) { // Every 5 seconds
    char buffer[120];
    sprintf(buffer, "PID Details - Error: %.1f, P: %.1f, I: %.1f, D: %.1f,
Out: %.1f",
            error, pTerm, iTerm, dTerm, output);
    Serial.println(buffer);
    lastDebugTime = now;
  }

  // Save values for next computation
  lastError = error;
  lastTime = now;

  return output;
}
```

serial_monitor.h

```
#ifndef SERIAL_MONITOR_H
#define SERIAL_MONITOR_H

#include <Arduino.h>
#include <stdio.h>
```

```
// Initialize serial communication with stdio support
void SerialMonitor_Init();

// Print a message to serial
void SerialMonitor_PrintMessage(const char* message);

// Report data for plotting
void SerialMonitor_ReportData(int setpoint, int rpm, int pwm);

// Setup FILE streams for stdin and stdout
extern FILE uart_output;
extern FILE uart_input;

#endif
```

serial_monitor.cpp

```
#include "serial_monitor.h"
#include <Arduino.h>
#include <stdio.h>

#define SERIAL_BAUD_RATE 9600

// Setup FILE streams for stdin and stdout
FILE uart_output;
FILE uart_input;
```

```
// Function to redirect printf to Serial
int uart_putchar(char c, FILE *stream) {
  Serial.write(c);
  return 0;
}

// Function to redirect stdin to Serial
int uart_getchar(FILE *stream) {
  while (!Serial.available());
  return Serial.read();
}

void SerialMonitor_Init() {
  // Initialize serial communication
  Serial.begin(SERIAL_BAUD_RATE);
  while (!Serial) {
    ; // Wait for serial port to connect
  }

  // Initialize stdio
  fdev_setup_stream(&uart_output, uart_putchar, NULL, _FDEV_SETUP_WRITE);
  fdev_setup_stream(&uart_input, NULL, uart_getchar, _FDEV_SETUP_READ);
  stdout = &uart_output;
  stdin = &uart_input;

  delay(100);
}
```

```
void SerialMonitor_PrintMessage(const char* message) {
  printf("%s\n", message);
}

void SerialMonitor_ReportData(int setpoint, int rpm, int pwm) {
  // Format for Arduino Serial Plotter
  printf("SetPoint:%d,RPM:%d,PWM:%d\n", setpoint, rpm, pwm);
}
```

tachometer.h

```
#ifndef TACHOMETER_H
#define TACHOMETER_H

// Initialize tachometer
void Tachometer_Init(int tachPin);

// Get current RPM
int Tachometer_GetRPM();

#endif
```

tachometer.cpp

```
#include "tachometer.h"
#include <Arduino.h>
```

```
// Constants
#define FILTER_SIZE 5 // Size of moving average filter
#define FAN_POLES 2   // Most PC fans have 2 poles (2 pulses per revolution)

// Variables
static int tachometerPin = 2;  // Default tachometer pin
static int currentRPM = 0;
static volatile unsigned long pulseCount = 0;
static unsigned long lastMeasurementTime = 0;
static const unsigned long measurementInterval = 1000; // 1 second

// Filter variables
static int rpmReadings[FILTER_SIZE];
static int readIndex = 0;
static int rpmTotal = 0;

// Interrupt handler for tachometer pulses
void countPulse() {
  pulseCount++;
}

void Tachometer_Init(int tachPin) {
  tachometerPin = tachPin;
  pinMode(tachometerPin, INPUT_PULLUP);

  // Initialize the moving average filter
  for (int i = 0; i < FILTER_SIZE; i++) {
```

```
    rpmReadings[i] = 0;
  }
  rpmTotal = 0;
  readIndex = 0;

  // Attach interrupt to count pulses
  attachInterrupt(digitalPinToInterrupt(tachometerPin), countPulse,
FALLING);

  printf("Tachometer initialized on pin %d\n", tachometerPin);
}

int Tachometer_GetRPM() {
  unsigned long currentTime = millis();

  // Update RPM measurement every measurementInterval
  if (currentTime - lastMeasurementTime >= measurementInterval) {
    // Calculate RPM: (pulses * 60) / (time in seconds * poles)
    // Detach interrupt temporarily to avoid race conditions
    detachInterrupt(digitalPinToInterrupt(tachometerPin));

    // Calculate RPM
    int rpm = (pulseCount * 60) / (FAN_POLES);

    // Reset pulse counter
    pulseCount = 0;

    // Reattach interrupt
```

```
    attachInterrupt(digitalPinToInterrupt(tachometerPin), countPulse,
FALLING);


    // Update filter
    rpmTotal = rpmTotal - rpmReadings[readIndex];
    rpmReadings[readIndex] = rpm;
    rpmTotal = rpmTotal + rpm;
    readIndex = (readIndex + 1) % FILTER_SIZE;


    // Calculate filtered RPM
    currentRPM = rpmTotal / FILTER_SIZE;


    lastMeasurementTime = currentTime;
  }


  return currentRPM;
}
```

user_interface.h

```
#ifndef USER_INTERFACE_H
#define USER_INTERFACE_H


// Initialize user interface
void UserInterface_Init();


// Update user interface (check for inputs)
void UserInterface_Update();
```

```
#endif
```

user_interface.cpp

```
#include "user_interface.h"
#include "pid_control.h"
#include "fan_control.h"
#include "tachometer.h"
#include <Arduino.h>
#include <stdio.h>

#define MAX_RPM 3000
#define MIN_RPM 500
#define RPM_STEP 100
#define COMMAND_BUFFER_SIZE 10

// Buffer for parsing commands
static char commandBuffer[COMMAND_BUFFER_SIZE];
static int bufferPos = 0;

void UserInterface_Init() {
  // Initialize built-in LED for alarm
  pinMode(LED_BUILTIN, OUTPUT);

  // Display initial help message
  printf("\n--- Fan Control System ---\n");
```

```
  printf("Available commands:\n");
  printf("  + : Increase target RPM by 100\n");
  printf("  - : Decrease target RPM by 100\n");
  printf("  rXXXX : Set target RPM to XXXX (e.g., r2000)\n");
  printf("  d : Run diagnostic test\n");
  printf("  p : Print current parameters\n");
  printf("  h : Show this help message\n");
  printf("-----------------------------------------------\n");
}

void UserInterface_Update() {
  // Check for serial input to adjust setpoint
  while (Serial.available() > 0) {
    char input = Serial.read();

    // Simple single-character commands
    if (input == '+' || input == '-' || input == 'd' || input == 'D' ||
input == 'p' || input == 'P' || input == 'h' || input == 'H') {
      // Reset command buffer
      bufferPos = 0;
      memset(commandBuffer, 0, COMMAND_BUFFER_SIZE);

      // Current setpoint
      double currentSetpoint = PIDControl_GetSetpoint();

      // Adjust setpoint based on input
      if (input == '+') {
        // Increase setpoint
```

```
        double newSetpoint = min(currentSetpoint + RPM_STEP,
(double)MAX_RPM);
        PIDControl_SetSetpoint(newSetpoint);

        // Notify user
        printf("New setpoint: %.0f RPM\n", newSetpoint);
      }
      else if (input == '-') {
        // Decrease setpoint
        double newSetpoint = max(currentSetpoint - RPM_STEP,
(double)MIN_RPM);
        PIDControl_SetSetpoint(newSetpoint);

        // Notify user
        printf("New setpoint: %.0f RPM\n", newSetpoint);
      }
      else if (input == 'd' || input == 'D') {
        // Run diagnostic test - slowly ramp PWM up and down
        printf("Starting diagnostic test...\n");

        // Save current PID output
        int savedPWM = FanControl_GetSpeed();

        // Ramp up slowly
        printf("Ramping PWM up from 0 to 255:\n");
        for (int pwm = 0; pwm <= 255; pwm += 10) {
          FanControl_SetSpeed(pwm);
          delay(300);  // Give time for fan to respond
```

```
      int rpm = Tachometer_GetRPM();
      printf("PWM: %d, RPM: %d\n", pwm, rpm);
    }

    delay(1000);  // Hold at max

    // Ramp down slowly
    printf("Ramping PWM down from 255 to 0:\n");
    for (int pwm = 255; pwm >= 0; pwm -= 10) {
      FanControl_SetSpeed(pwm);
      delay(300);  // Give time for fan to respond
      int rpm = Tachometer_GetRPM();
      printf("PWM: %d, RPM: %d\n", pwm, rpm);
    }

    // Restore original PWM
    printf("Diagnostic test complete, resuming normal operation\n");
    FanControl_SetSpeed(savedPWM);
  }
  else if (input == 'p' || input == 'P') {
    // Print current parameters
    printf("Current system parameters:\n");
    printf("Setpoint: %.0f RPM, Current RPM: %d, PWM: %d\n",
           PIDControl_GetSetpoint(), Tachometer_GetRPM(),
FanControl_GetSpeed());
  }
  else if (input == 'h' || input == 'H') {
    // Show help message
```

```
    printf("\n--- Fan Control System ---\n");
    printf("Available commands:\n");
    printf("  + : Increase target RPM by 100\n");
    printf("  - : Decrease target RPM by 100\n");
    printf("  rXXXX : Set target RPM to XXXX (e.g., r2000)\n");
    printf("  d : Run diagnostic test\n");
    printf("  p : Print current parameters\n");
    printf("  h : Show this help message\n");
    printf("-----------------------------------------\n");
  }
}
// Command that requires parsing (r for direct RPM setting)
else {
  // Add to buffer if not newline or carriage return
  if (input != '\n' && input != '\r') {
    if (bufferPos < COMMAND_BUFFER_SIZE - 1) {
      commandBuffer[bufferPos++] = input;
      commandBuffer[bufferPos] = '\0';
    }
  }
  // Process the command when newline is received
  else if (bufferPos > 0) {
    // Process RPM setting command
    if (commandBuffer[0] == 'r' || commandBuffer[0] == 'R') {
      // Parse the number after 'r'
      int targetRpm = atoi(commandBuffer + 1);
```

```
          // Ensure it's within valid range
          if (targetRpm >= MIN_RPM && targetRpm <= MAX_RPM) {
            PIDControl_SetSetpoint(targetRpm);
            printf("Target RPM set to: %d\n", targetRpm);
          } else {
            printf("Invalid RPM value. Please use a value between %d and %d\n", MIN_RPM, MAX_RPM);
          }
        }

        // Reset buffer after processing
        bufferPos = 0;
        memset(commandBuffer, 0, COMMAND_BUFFER_SIZE);
      }
    }
  }
}
```

main.cpp

```
#include <Arduino.h>
#include <stdio.h>
#include "pid_control.h"
#include "fan_control.h"
#include "tachometer.h"
#include "serial_monitor.h"
#include "user_interface.h"
```

```
// Pin definitions
#define PWM_PIN 9      // Blue wire (PWM)
#define TACH_PIN 2     // Yellow wire (tachometer)
#define ALARM_LED_PIN LED_BUILTIN

// PID and fan control parameters
#define INITIAL_SETPOINT 1500    // Initial target RPM
#define INITIAL_PWM_VALUE 128    // Initial PWM value (50%)
#define REPORT_INTERVAL 500      // Report data every 500ms
#define MAX_RPM_DEVIATION 300    // Maximum allowed RPM deviation before
alarm

void setup() {
  // Initialize serial communication with stdio support
  SerialMonitor_Init();
  printf("Fan PID Control System Started\n");

  // Set up LED pin for alarm
  pinMode(ALARM_LED_PIN, OUTPUT);

  // Initialize fan control with PWM
  FanControl_Init(PWM_PIN);

  // Initialize tachometer for RPM measurement
  Tachometer_Init(TACH_PIN);

  // Initialize PID controller with tuning parameters
  PIDControl_Init(0.5, 0.2, 0.05);  // kp, ki, kd
```

```
  PIDControl_SetSetpoint(INITIAL_SETPOINT);

  // Initialize user interface
  UserInterface_Init();

  // Start fan at 50% speed and wait for it to start
  printf("Setting initial fan speed to 50%%\n");
  FanControl_SetSpeed(INITIAL_PWM_VALUE);

  // Allow time for the fan to start
  delay(2000);
}

void loop() {
  // Get current time
  unsigned long currentTime = millis();

  // Handle user interface (setpoint adjustments)
  UserInterface_Update();

  // Read current RPM from tachometer
  int currentRPM = Tachometer_GetRPM();

  // Update PID controller with current RPM
  PIDControl_SetInput(currentRPM);
  double pidOutput = PIDControl_Compute();
```

```
  // Set fan speed based on PID output
  FanControl_SetSpeed((int)pidOutput);

  // Check for large deviation and set alarm if needed
  int setpoint = PIDControl_GetSetpoint();
  if (abs(currentRPM - setpoint) > MAX_RPM_DEVIATION) {
    // Deviation too large, set alarm
    digitalWrite(ALARM_LED_PIN, HIGH);
  } else {
    digitalWrite(ALARM_LED_PIN, LOW);
  }

  // Report data periodically
  static unsigned long lastReportTime = 0;
  if (currentTime - lastReportTime >= REPORT_INTERVAL) {
    SerialMonitor_ReportData(setpoint, currentRPM, (int)pidOutput);
    lastReportTime = currentTime;
  }
}
```