Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

Report

Laboratory work nr. 5.1

Control funcțional - ***ON-OFF cu Histereza***

Embedded Systems

Prepared by:

Berco Andrei, FAF-221

Checked by:

Martîniuc Alexei, *university assistant*

Chișinău – 2025

1. OBJECTIVES

Main purpose of the work:

Developing a modular application for a microcontroller (MCU) that implements an ON-OFF control system with hysteresis for a chosen physical parameter (e.g., temperature, humidity, or rotor position). The system will be actuated using a relay or L298 driver, with the set point adjustable via a selected input method and the current state displayed on an LCD and/or serial interface using STDIO.

The objectives of the work:

- Understanding the principles of ON-OFF control with hysteresis.
- Applying hysteresis in control systems involving relays or L298 drivers.
- Designing and implementing a modular embedded application with LCD and STDIO interfaces.
- Displaying and adjusting control parameters (set point, measured value) in real time.
- Structuring the software using separate modules for each peripheral to promote reusability.
- Providing architectural diagrams and electrical schematics to support the design.

PROBLEM DEFINITION

Develop an MCU-based application that implements an ON-OFF control system with hysteresis for one of the following options:

- **Variant A**: Control of temperature or humidity using data from a digital sensor (e.g., DHT22), actuated through a relay.
- **Variant B**: Rotor position control using a servo motor driven via an L298 driver, applying ON-OFF hysteresis logic. The motor operates in saturation mode (fixed 50% power) and changes direction depending on the control deviation.

For both variants:

- The reference value (Set Point) should be adjustable via one of the following: potentiometer, UP/DOWN buttons, encoder, 4x4 keypad, or serial interface.
- The Set Point and current measured value should be displayed on a 2x16 or 4x20 LCD and/or

via the serial terminal using printf.

- A configurable hysteresis value (fixed or defined in code) must be used to prevent frequent switching.
- The control logic follows the rule: ON below the lower threshold, OFF above the upper threshold

2. DOMAIN ANALYSIS

1. Technology Stack

To complete this project, the following hardware components were used:

- Microcontroller (ESP32 / STM32 / Arduino UNO with STDIO support)
- Digital sensor (e.g., DHT22 for temperature or humidity, or potentiometer/encoder for position)
- Actuator (Relay module or L298N motor driver with motor)
- LCD display (2x16 or 4x20, preferably I2C)
- Input devices (4x4 keypad, UP/DOWN buttons, encoder, potentiometer)
- Breadboard, jumper wires, resistors
- Power supply (USB or external, depending on actuator requirements)

Software environment:

- IDE: Visual Studio Code with PlatformIO extension
- Libraries: STDIO (Serial), LCD display driver (LiquidCrystal_I2C or similar), sensor libraries (e.g., DHT.h)
- Serial tools: Monitor Serial (PlatformIO), TeraTerm or Putty for command and feedback
- Optional: Proteus simulator for hardware emulation and validation

2. Use Cases

ON-OFF control with hysteresis is widely used in systems where binary actuation is required and noise or small fluctuations could lead to unnecessary switching.

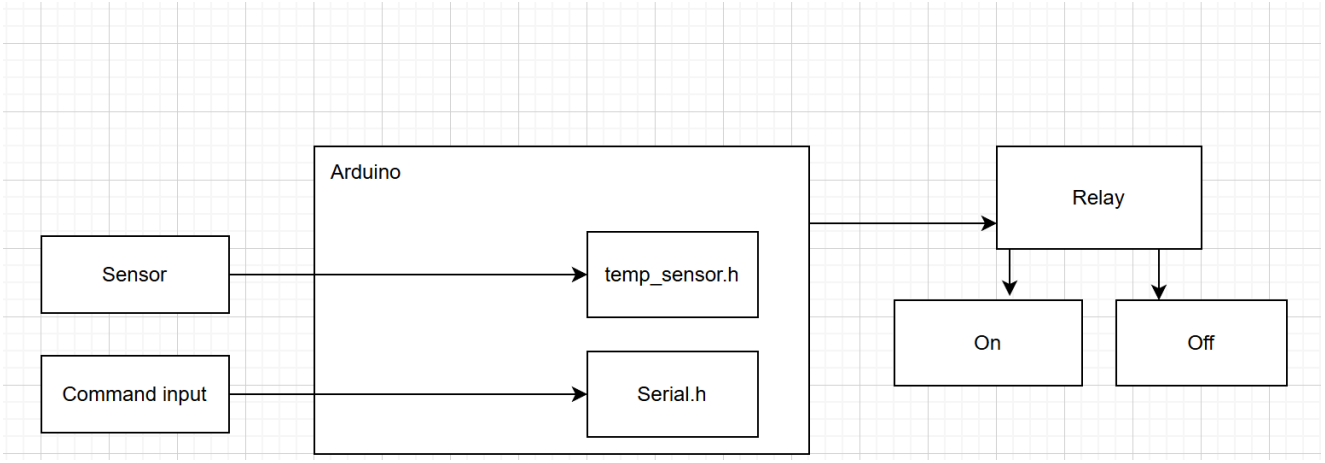
- Temperature control systems: For instance, in a smart thermostat, a heater is turned ON when the temperature drops below a threshold and OFF once it exceeds a higher threshold, avoiding rapid toggling around a single set point.
- Humidity regulation: Automatically activating a humidifier or dehumidifier when readings cross specific bounds, maintaining a stable indoor climate.
- Position control systems: In robotics or CNC machines, the system ensures a component stays within a tolerance window without jittering due to small deviations.
- Ventilation or fan control: Based on temperature/humidity, fans can be toggled ON/OFF with hysteresis to balance comfort and energy efficiency.

Architectural Design

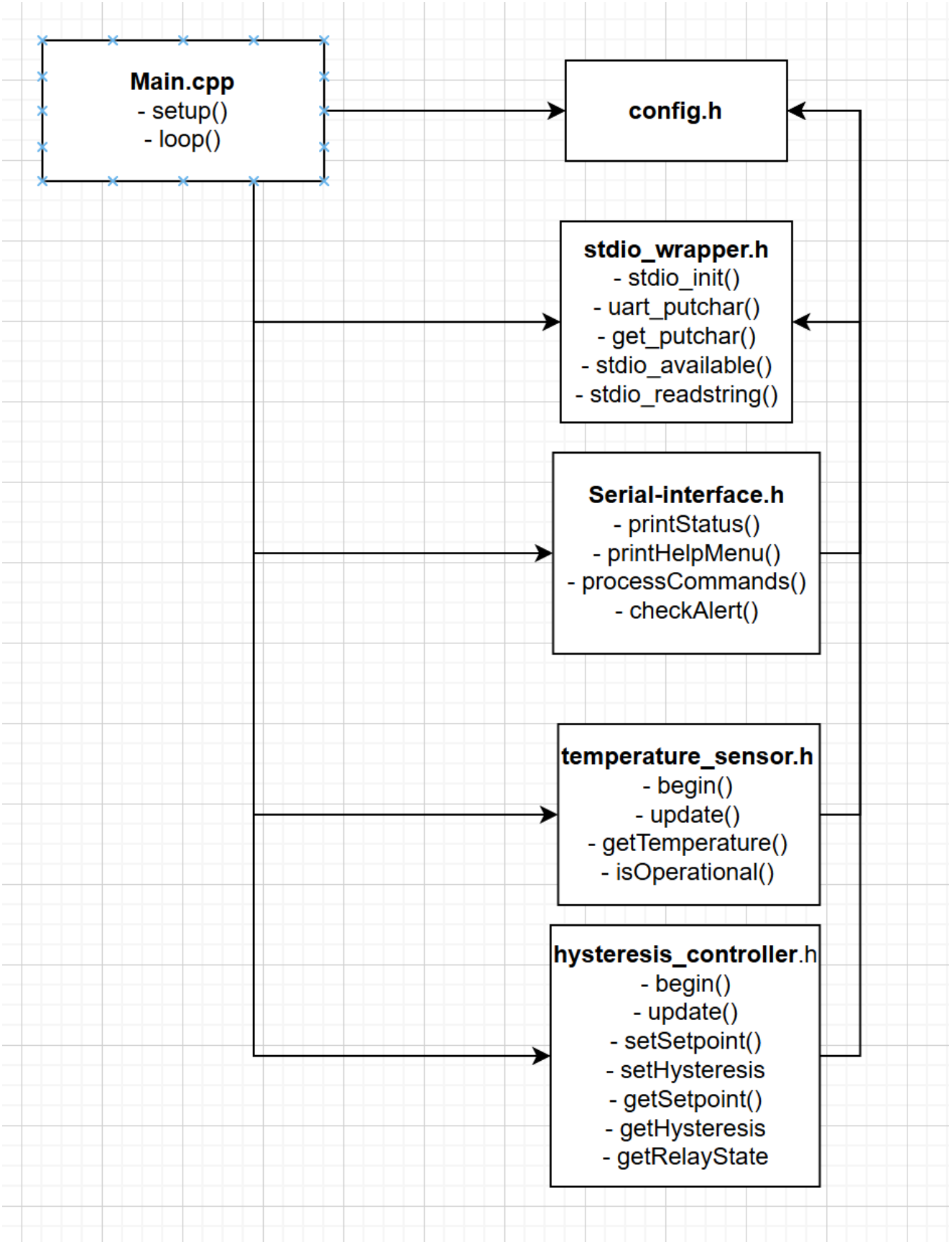
The architectural design emphasizes modularity, reusability, and real-time interaction. Each component of the system is encapsulated in its own module:

- **MCU Core**: Manages overall logic, reads sensor input, processes control algorithms, and updates outputs.
- **Sensor Module**: Reads environmental or positional data (e.g., DHT22 or potentiometer), returning processed values.
- **Control Module**: Implements ON-OFF logic with hysteresis, calculates thresholds, and determines output state.
- **Actuator Module**: Controls relay or motor via L298 driver based on control output.
- **Interface Module**: Displays current and set values on LCD and allows interaction via serial (STDIO) or physical controls (buttons/keypad).

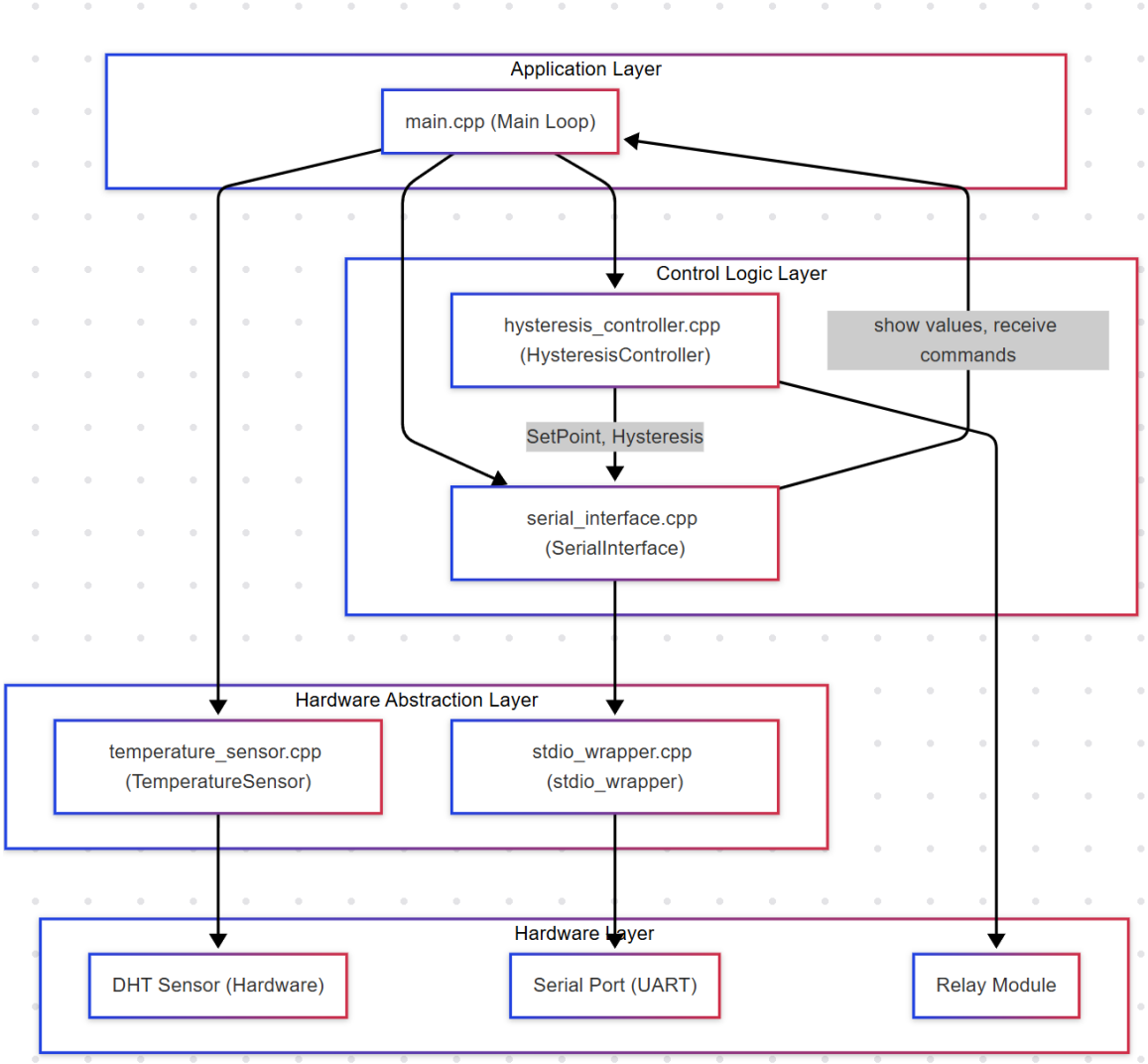
Hardware Diagram



Serial receives commands from user. Temp_sensor receives data from the temperature sensor. According to the serial command, the hysteresis data is updated. Next, according to temp data, the relay can change its state, either on or off, and to serial is printed a message.

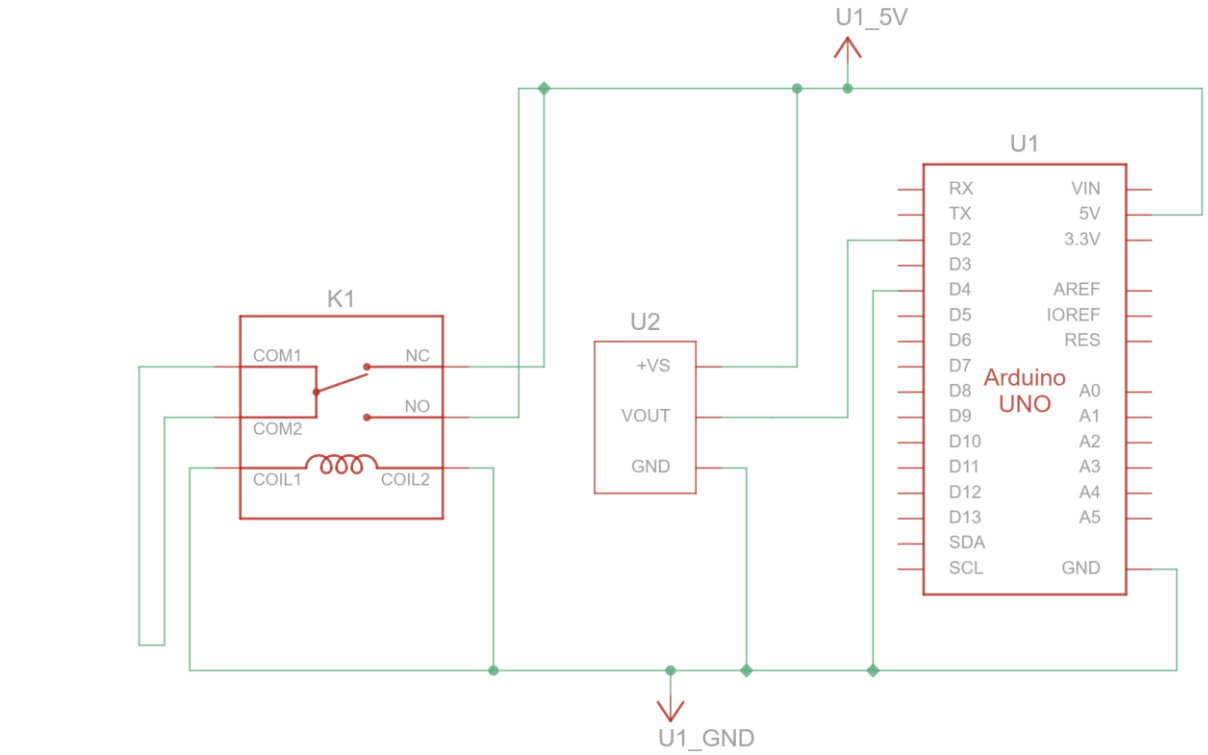

Main.cpp
- setup()
- loop()
config.h
stdio_wrapper.h
- stdio_init()
- uart_putchar()
- get_putchar()
- stdio_available()
- stdio_readstring()
Serial-interface.h
- printStatus()
- printHelpMenu()
- processCommands()
- checkAlert()
temperature_sensor.h
- begin()
- update()
- getTemperature()
- isOperational()
hysteresis_controller.h
- begin()
- update()
- setSetpoint()
- setHysteresis
- getSetpoint()
- getHysteresis
- getRelayState

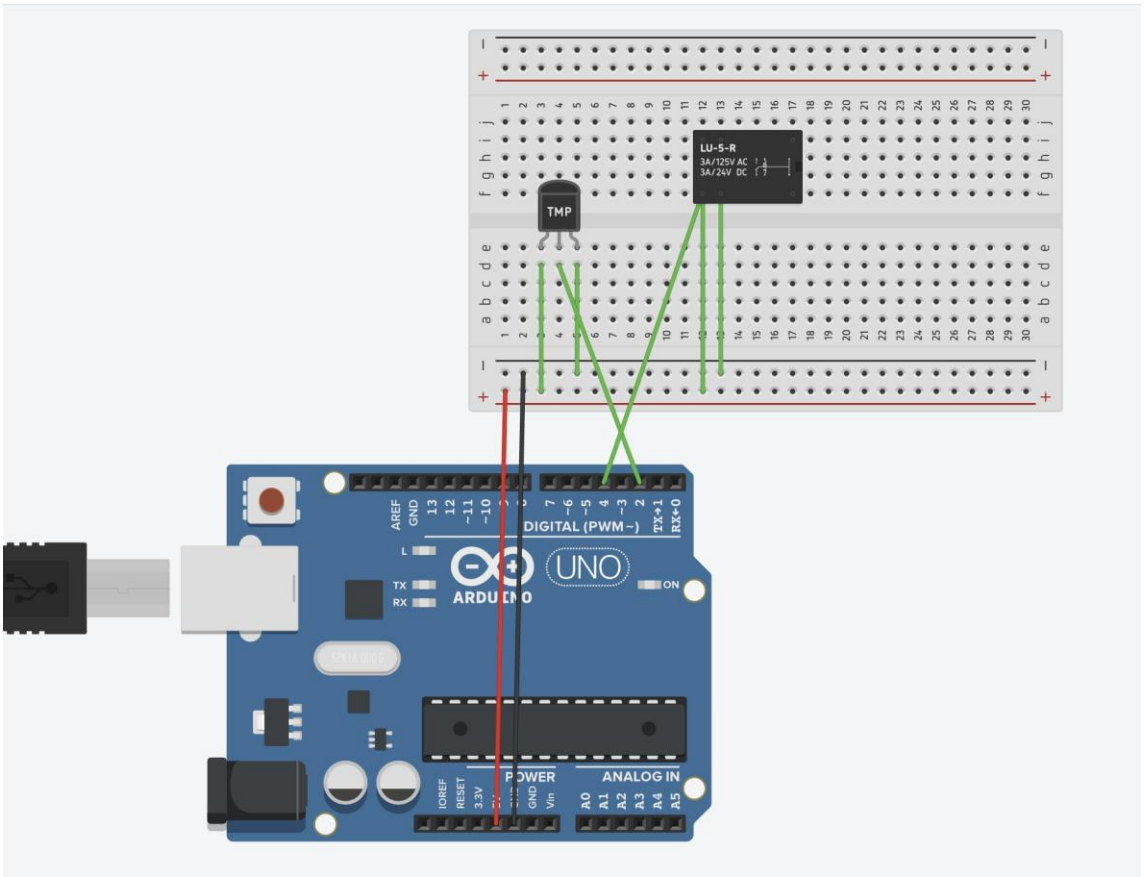
Layer Diagram



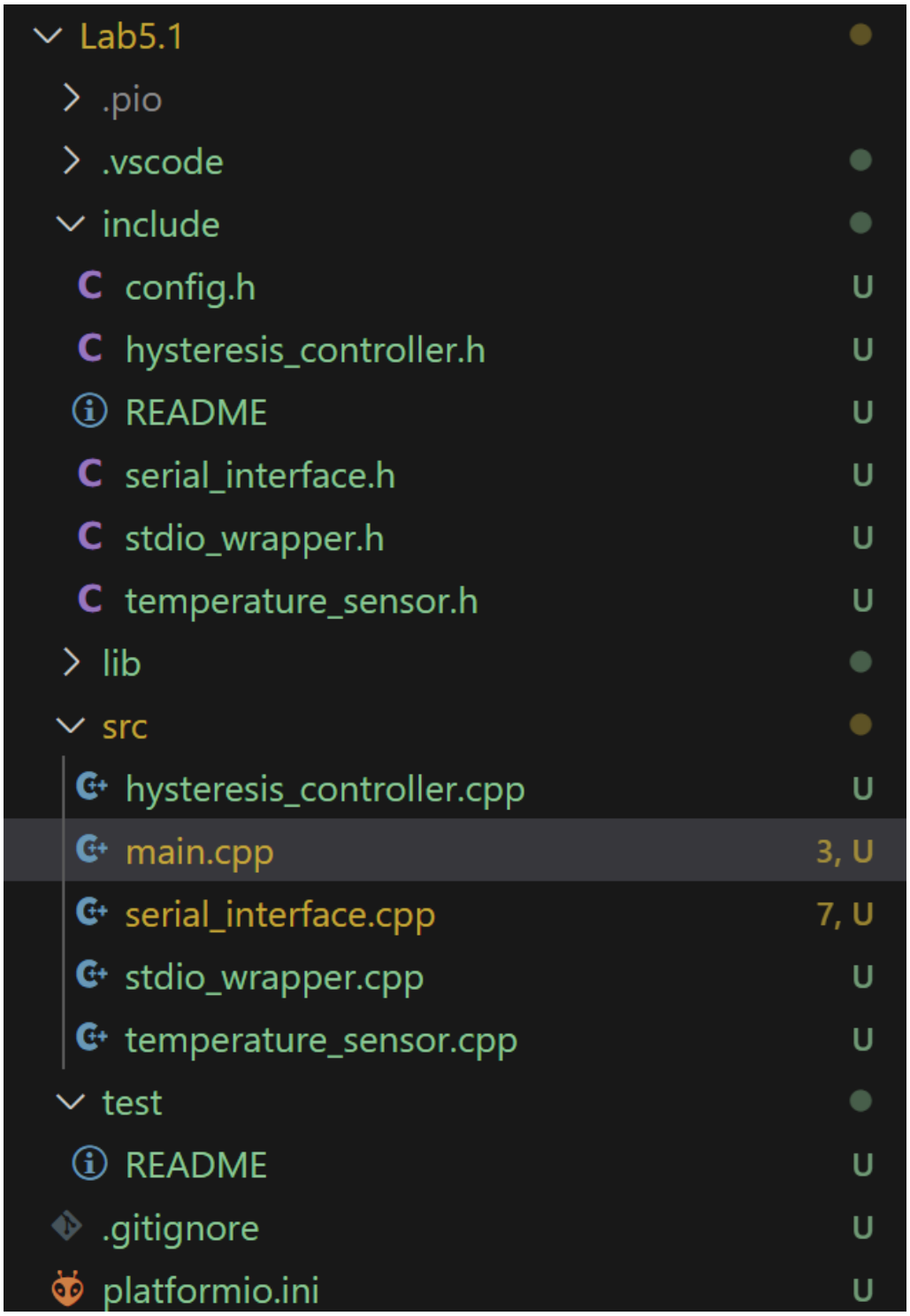
- **Application Layer**: Your main loop, coordinating everything.
- **Control Logic Layer**: Decision-making components (hysteresis control + serial command UI).
- **Hardware Abstraction Layer**: Interfaces with sensors and serial using software.
- **Hardware Layer**: Physical components (sensor, relay, serial UART).

Electrical Scheme

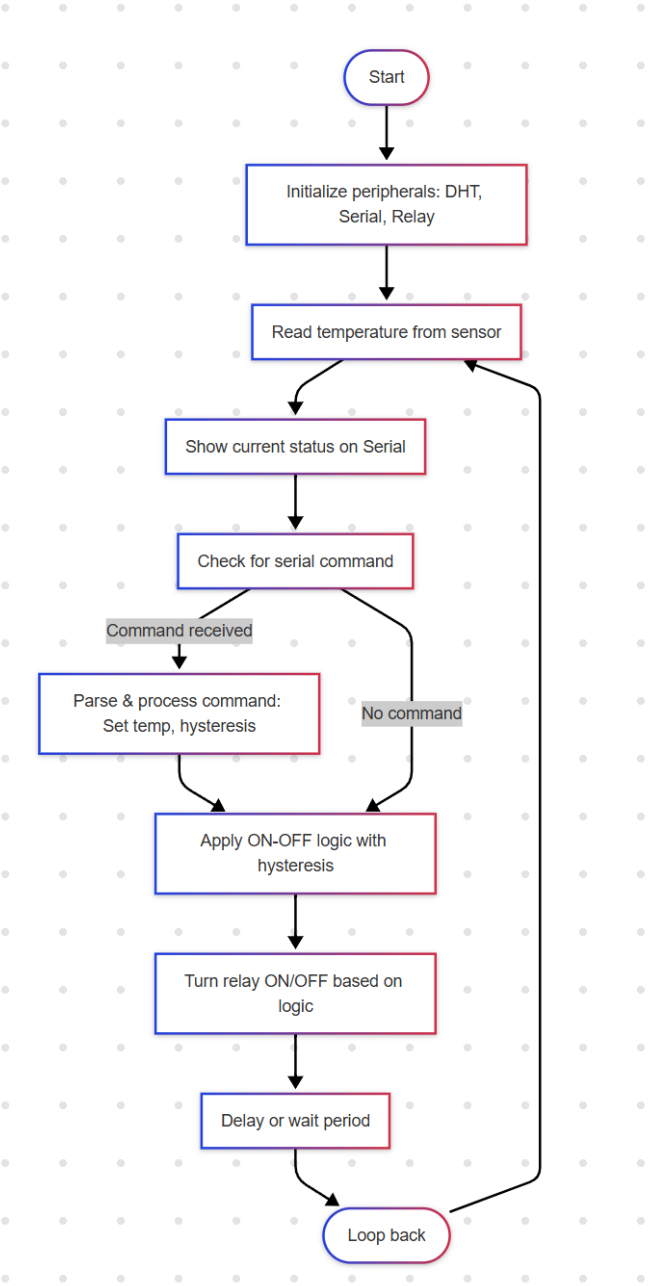




Project Structure and Modular Implementation



Block Diagram of System Behaviour



The **flowchart** outlines the **overall control loop** of your temperature control system running on a microcontroller:

1. **Start / Initialization**:

   The system begins by initializing all peripherals like the temperature sensor (DHT), serial communication, and relay control.

2. **Read Temperature**:

   It continuously reads the current temperature from the sensor.

3. **Display Status**:

   The temperature and relay status are printed to the serial monitor for user feedback.

4. **Command Check**:

   The system checks if a user has entered any command via serial input.

5. **Process Command**:

   If a command is received (e.g., setting a new temperature threshold or adjusting hysteresis), it parses and applies the changes.

6. **Apply ON-OFF Logic**:

   Using the current temperature, setpoint, and hysteresis, the system decides whether to turn the relay on or off.

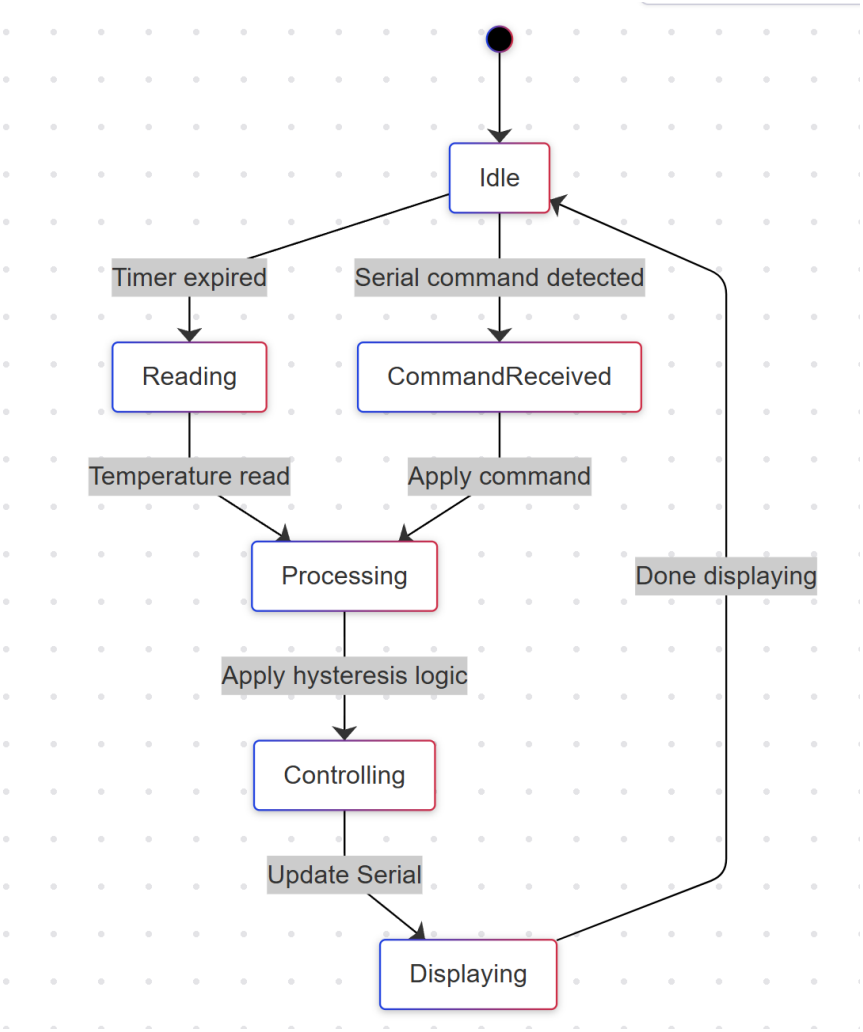
7. **Control Relay**:

   It sends the actual ON/OFF command to the relay connected to a heating or cooling device.

8. **Delay / Looping**:

   A short delay is introduced to prevent rapid switching, then the loop restarts.

Functional Block Diagrams



Description:

The **FSM diagram** models the system as a sequence of **states** with well-defined transitions:

1. **Idle**:

   The system waits in a passive state, either for a timer tick (to recheck temperature) or for a user command.

2. **Reading**:

   Triggered by a timer or scheduled interval, the system reads the temperature sensor.

3. **Processing**:

   It evaluates the sensor data and/or processes any new command received from the user.

4. **Controlling**:

   Applies the control logic (e.g., whether to turn the relay ON or OFF based on hysteresis logic).

5. **Displaying**:

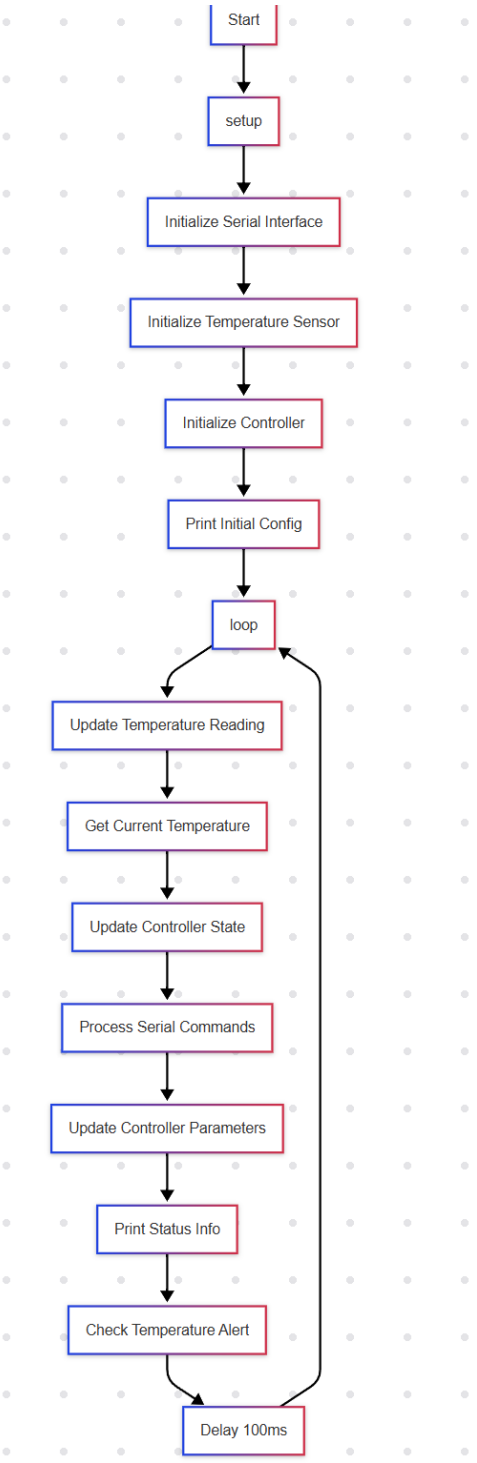
   Outputs system status to the serial monitor or user interface.

6. **CommandReceived**:

If a command is detected in Idle, the system temporarily enters this state to prioritize parsing and applying user input before returning to normal processing.

Code flowchart:

Main.cpp

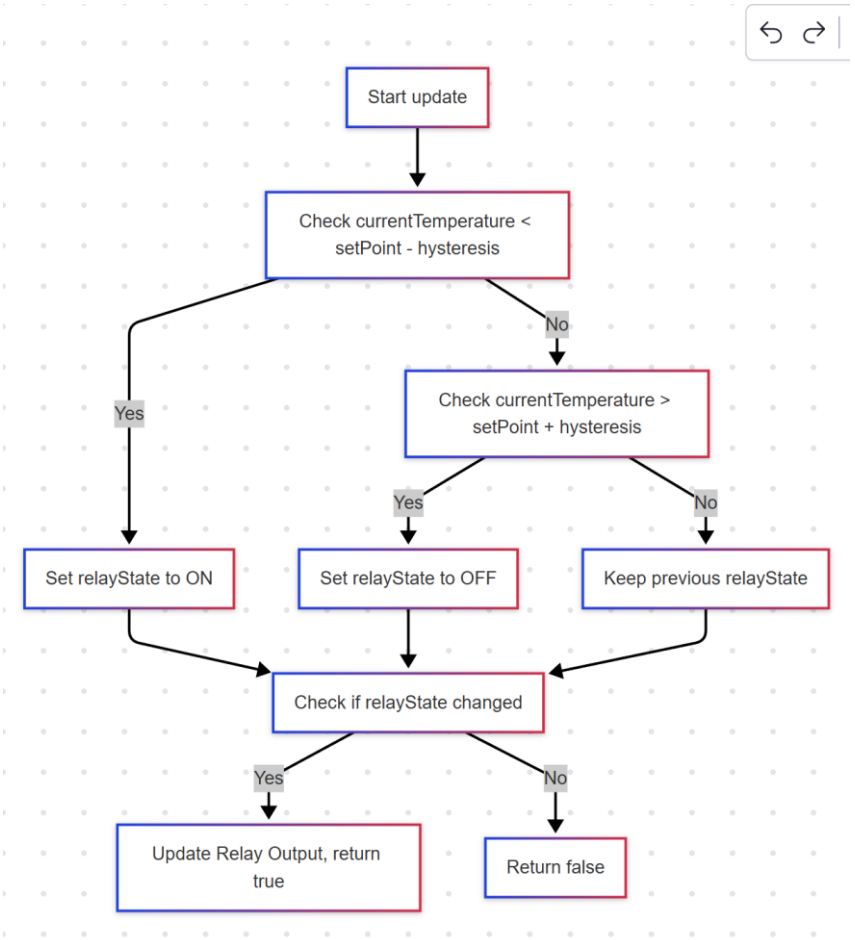


Description:

This is the **entry point** of the application. It initializes the temperature sensor, serial interface, and hysteresis controller. Inside the loop(), it continuously:

- Updates temperature,
- Applies control logic,
- Responds to serial commands,
- Prints system status,
- Checks for abnormal temperature deviations.

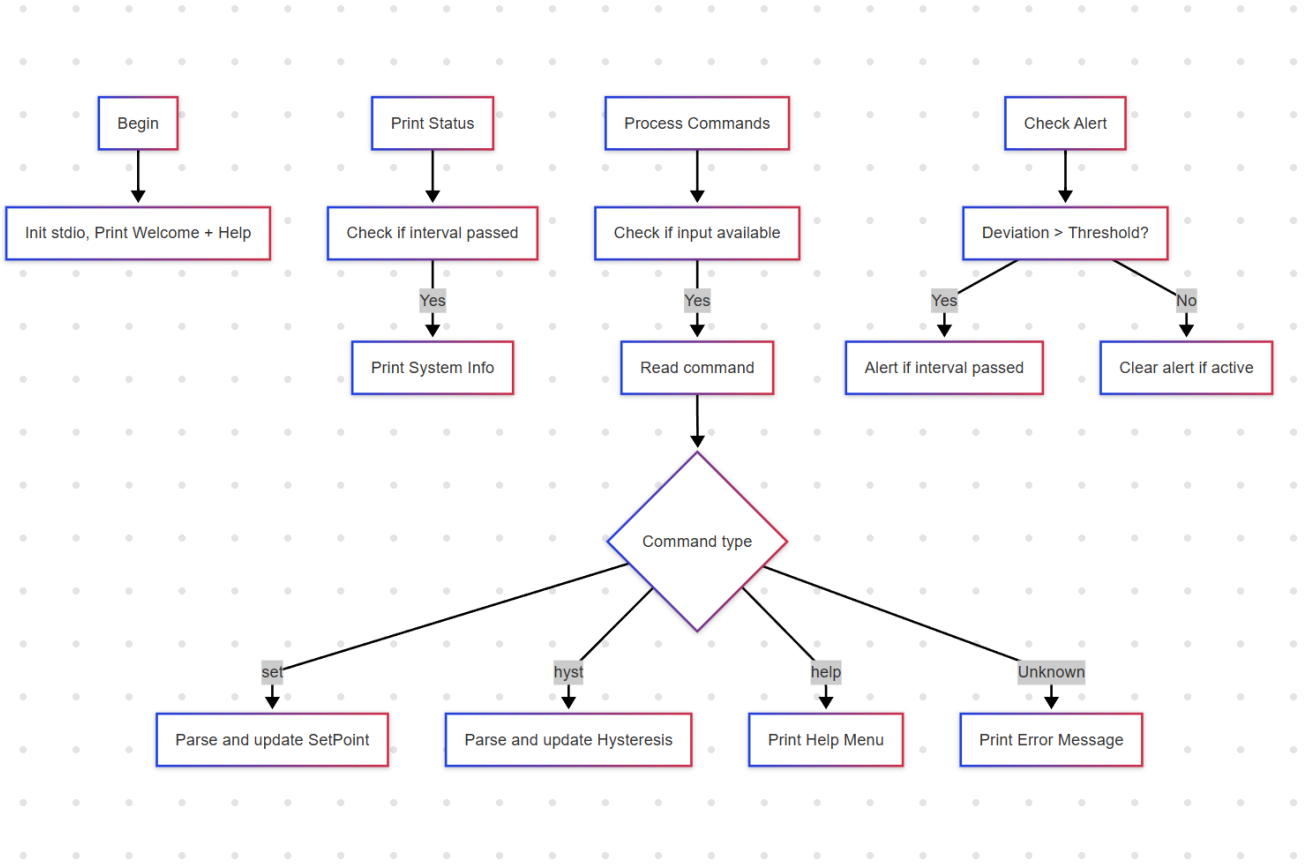
hysteresis_controller.cpp



Description:

This module implements the **ON/OFF hysteresis control logic**. It turns a relay ON or OFF depending on whether the temperature is outside the defined threshold range around the setpoint.

Serial_interface.cpp



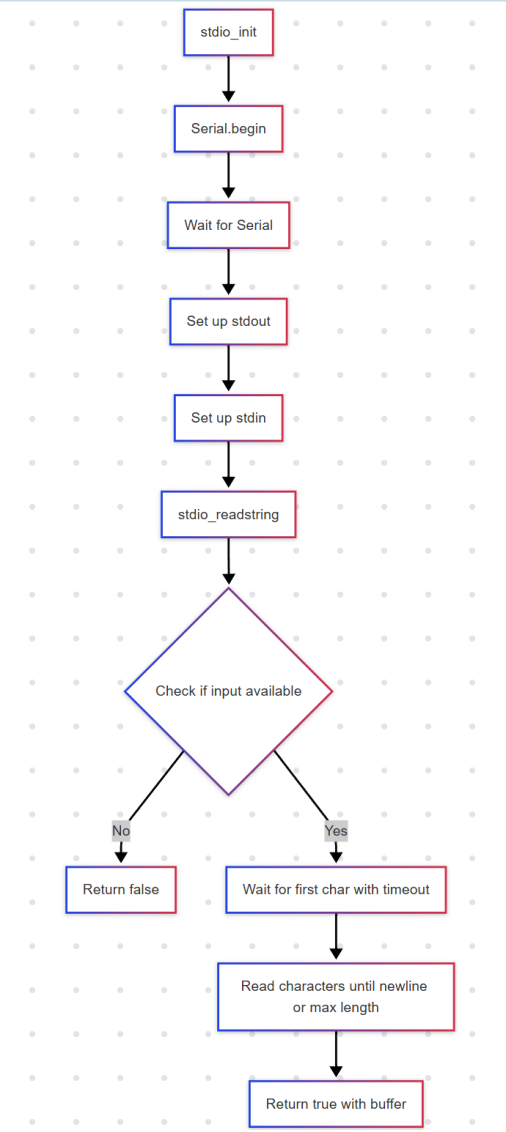
Description:

Handles user interaction over serial communication using stdio. It:

- Displays a help menu,
- Accepts and processes commands like setting the setpoint and hysteresis,
- Periodically prints temperature and control status,
- Issues alerts when temperature deviates significantly.

\

Stdio_wrapper.cpp



Wraps Arduino's Serial I/O in stdio functionality. It allows using printf, scanf, etc., and handles both reading and writing character streams. It includes a function to read user input into a buffer.

Conclusion

In this lab, we successfully designed and implemented a **Temperature Control System with ON-OFF Hysteresis** using Arduino. The project involved integrating several core components: a temperature sensor, a hysteresis controller, and a serial communication interface for user interaction. Each module was carefully crafted and validated to ensure reliable performance in a temperature regulation scenario.

Key Highlights:

1. **Temperature Sensor Integration:** The TemperatureSensor class interfaces with the DHT

sensor and successfully reads temperature data at predefined intervals. The system ensures accurate readings through the initialization and periodic updates of the sensor state, allowing the controller to operate effectively. The sensor's responsiveness and the method for checking its operational status were crucial for reliable performance.

2. **Hysteresis Control Logic:** The HysteresisController class implements ON-OFF control with hysteresis, a technique commonly used in simple thermostats to prevent rapid cycling of the relay. The controller uses a setpoint and hysteresis values to determine whether the relay should be turned ON or OFF based on the current temperature. The logic was implemented to ensure smooth operation by turning the relay ON when the temperature falls below the lower threshold (setpoint - hysteresis) and OFF when the temperature exceeds the upper threshold (setpoint + hysteresis). This approach eliminates frequent switching that could otherwise result from small temperature fluctuations.
3. **Serial Interface for User Interaction:** The SerialInterface class provided a means for the user to interact with the system through the serial terminal. It allows for real-time updates of the system's status, such as the current temperature, setpoint, and relay state. Furthermore, it provides commands to adjust the setpoint and hysteresis value, enabling the user to modify the system's behavior dynamically. This flexibility makes the system adaptable to various temperature regulation needs.
4. **System Alerts:** A noteworthy feature added to the system was the **temperature deviation alert**. When the temperature exceeds a predefined threshold from the setpoint, an alert is triggered, notifying the user of potential issues. This feature enhances the robustness of the system by allowing for timely interventions when the temperature deviates too far from the desired range.
5. **Modular and Scalable Design:** The system's modular approach, with separate files for different components (sensor, controller, serial interface, and I/O wrapper), ensures scalability and maintainability. Each component is independently responsible for a specific task, reducing complexity and making future updates easier. This separation also promotes code reusability, as the individual modules could be adapted for different types of control systems with minimal changes.
6. **User-Friendliness:** The use of the serial interface for user commands (e.g., setting the temperature or hysteresis) is an effective way to interact with the system. The implementation of helpful messages and clear feedback, such as alert notifications and command descriptions, ensures that users can easily operate and troubleshoot the system.

Challenges:

While the implementation was generally successful, there were a few challenges faced during the lab work:

- **Sensor Calibration:** Ensuring the temperature sensor was correctly calibrated was initially tricky. However, by verifying its response and accounting for potential measurement inaccuracies, the system was able to maintain reliable temperature readings.
- **Debouncing User Inputs:** Managing user inputs via the serial interface required careful handling to avoid erroneous inputs. Adequate command parsing and validation helped mitigate this issue.

Future Improvements:

- **Enhanced Precision:** To improve the control accuracy, a more advanced sensor with higher resolution could be used, and additional feedback mechanisms could be implemented to fine-tune the relay switching.
- **Wi-Fi Integration:** Future versions could incorporate wireless communication (e.g., using Wi-Fi or Bluetooth) to allow remote control and monitoring of the system.
- **Advanced Control Algorithms:** More sophisticated control strategies, such as proportional-integral-derivative (PID) control, could be implemented to provide finer control over the temperature and minimize fluctuations.

Final Thoughts:

This lab provided valuable hands-on experience in both hardware (sensor and relay integration) and software (control algorithms and serial communication). It demonstrated the utility of hysteresis in temperature control systems and gave insight into designing user-friendly interfaces for embedded systems. The modular approach adopted here is applicable to a wide range of embedded control systems, and the skills gained will be valuable for future projects in embedded systems and control engineering.

During the preparation of this report, the author used Grok for generating/strengthening the content. The resulting information was reviewed, validated, and adjusted according to the requirements of the laboratory work.

References

1. IoT-24 : Lab 5.1. - https://www.youtube.com/watch?v=ODJjs82LrtM
2. Arduino Documentation about Seria - https://docs.arduino.cc/language-reference/en/functions/communication/serial/

3. PlatformIO Documentation - https://docs.platformio.org/en/latest/integration/ide/pioide.html

Appendix - Source Code

Main.cpp

```
/**
 * main.cpp
 * Temperature control system with ON-OFF hysteresis using stdio
 */

#include <Arduino.h>
#include <stdio.h>
#include "config.h"
#include "stdio_wrapper.h"
#include "temperature_sensor.h"
#include "hysteresis_controller.h"
#include "serial_interface.h"

// Create instances of modules
TemperatureSensor tempSensor;
HysteresisController controller;
SerialInterface serialUI;

void setup() {
    // Initialize serial interface
    serialUI.begin();

    // Initialize temperature sensor
    printf("Initializing temperature sensor... ");
    if (tempSensor.begin()) {
        printf("OK\n");
    } else {
        printf("FAILED\n");
        printf("Check sensor connection and restart\n");
    }

    // Initialize controller
    controller.begin();
    printf("Initializing controller... ");
    printf("OK\n");

    // Print initial configuration
    printf("\nInitial setpoint: %.2f °C\n", controller.getSetPoint());
    printf("Initial hysteresis: ±%.2f °C\n", controller.getHysteresis());
    printf("\nSystem running. Type 'help' for commands.\n");
}

void loop() {
    // Update temperature reading
    tempSensor.update();
```

```
    // Get current temperature
    float currentTemperature = tempSensor.getTemperature();

    // Update controller state
    bool stateChanged = controller.update(currentTemperature);

    // Process serial commands
    float currentSetPoint = controller.getSetPoint();
    float currentHysteresis = controller.getHysteresis();

    bool commandProcessed = serialUI.processCommands(&currentSetPoint, &currentHysteresis);

    // Update controller parameters if changed via serial
    if (commandProcessed) {
        controller.setSetPoint(currentSetPoint);
        controller.setHysteresis(currentHysteresis);
    }

    // Print status information
    serialUI.printStatus(
        currentTemperature,
        controller.getSetPoint(),
        controller.getHysteresis(),
        controller.getRelayState()
    );

    // Check for temperature deviation alert (bonus feature)
    serialUI.checkAlert(currentTemperature, controller.getSetPoint());

    // Small delay to reduce CPU usage
    delay(100);
}
```

Hysteresis_controller.h

```
/**
 * hysteresis_controller.cpp
 * Implementation of ON-OFF control with hysteresis
 */

#include "hysteresis_controller.h"

HysteresisController::HysteresisController() {
    setPoint = DEFAULT_SETPOINT;
    hysteresis = DEFAULT_HYSTERESIS;
    relayState = false;
}

void HysteresisController::begin() {
    pinMode(RELAY_PIN, OUTPUT);
    digitalWrite(RELAY_PIN, LOW);  // Initialize relay as OFF
}

bool HysteresisController::update(float currentTemperature) {
    bool previousState = relayState;
```

```
    // Hysteresis control logic
    // - Turn ON if temperature falls below (setPoint - hysteresis)
    // - Turn OFF if temperature rises above (setPoint + hysteresis)

    if (currentTemperature < (setPoint - hysteresis)) {
        relayState = true;
    } else if (currentTemperature > (setPoint + hysteresis)) {
        relayState = false;
    }
    // If temperature is between thresholds, maintain previous state

    // Update relay output if state changed
    if (previousState != relayState) {
        digitalWrite(RELAY_PIN, relayState ? HIGH : LOW);
        return true;  // State changed
    }

    return false;  // No change
}

bool HysteresisController::setSetPoint(float newSetPoint) {
    // Validate setpoint within allowed range
    if (newSetPoint >= MIN_TEMPERATURE && newSetPoint <= MAX_TEMPERATURE) {
        setPoint = newSetPoint;
        return true;
    }
    return false;
}

void HysteresisController::setHysteresis(float newHysteresis) {
    // Ensure hysteresis is positive
    hysteresis = abs(newHysteresis);
}

float HysteresisController::getSetPoint() {
    return setPoint;
}

float HysteresisController::getHysteresis() {
    return hysteresis;
}

bool HysteresisController::getRelayState() {
    return relayState;
}

float HysteresisController::getLowerThreshold() {
    return setPoint - hysteresis;
}

float HysteresisController::getUpperThreshold() {
    return setPoint + hysteresis;
}
```

Serial_interface.cpp

```
/**
 * serial_interface.cpp
 * Implementation of serial communication interface using stdio
 */

#include "serial_interface.h"

SerialInterface::SerialInterface() {
    lastPrintTime = 0;
    lastAlertTime = 0;
    alertActive = false;
}

void SerialInterface::begin() {
    // Initialize stdio wrapper
    stdio_init(SERIAL_BAUD_RATE);

    // Print welcome message and help menu
    printf("\n=== Temperature Control System with Hysteresis ===\n");
    printHelpMenu();
}

void SerialInterface::printStatus(float temperature, float setPoint, float hysteresis, bool relayState) {
    unsigned long currentTime = millis();

    // Check if it's time to update the serial output
    if (currentTime - lastPrintTime >= SERIAL_PRINT_INTERVAL) {
        lastPrintTime = currentTime;

        // Print status information using printf formatting
        printf("\n--- System Status ---\n");
        printf("Temperature: %.2f °C\n", temperature);
        printf("Set Point: %.2f °C\n", setPoint);
        printf("Hysteresis: ±%.2f °C\n", hysteresis);
        printf("Thresholds: %.2f to %.2f °C\n", setPoint - hysteresis, setPoint + hysteresis);
        printf("Relay: %s\n", relayState ? "ON" : "OFF");
        printf("--------------------\n");
    }
}

void SerialInterface::printHelpMenu() {
    printf("\n--- Available Commands ---\n");
    printf("set:XX.X    - Set temperature setpoint (e.g., 'set:25.5')\n");
    printf("hyst:X.X    - Set hysteresis value (e.g., 'hyst:2.0')\n");
    printf("help        - Display this help menu\n");
    printf("------------------------\n");
}

bool SerialInterface::processCommands(float* currentSetPoint, float* currentHysteresis) {
    // Maximum command length
    const size_t MAX_CMD_LENGTH = 32;
    char command[MAX_CMD_LENGTH];

    // Check if there's input available and read it
```

```
    if (stdio_available()) {
        if (stdio_readstring(command, MAX_CMD_LENGTH)) {
            // Process setpoint command
            if (strncmp(command, "set:", 4) == 0) {
                float newSetPoint;
                if (sscanf(command + 4, "%f", &newSetPoint) == 1) {
                    if (newSetPoint >= MIN_TEMPERATURE && newSetPoint <= MAX_TEMPERATURE) {
                        *currentSetPoint = newSetPoint;
                        printf("New setpoint: %.2f °C\n", newSetPoint);
                        return true;
                    } else {
                        printf("Error: Setpoint must be between %.1f and %.1f °C\n",
                               MIN_TEMPERATURE, MAX_TEMPERATURE);
                    }
                } else {
                    printf("Error: Invalid setpoint format\n");
                }
            }
            // Process hysteresis command
            else if (strncmp(command, "hyst:", 5) == 0) {
                float newHysteresis;
                if (sscanf(command + 5, "%f", &newHysteresis) == 1) {
                    if (newHysteresis > 0) {
                        *currentHysteresis = newHysteresis;
                        printf("New hysteresis: ±%.2f °C\n", newHysteresis);
                        return true;
                    } else {
                        printf("Error: Hysteresis must be positive\n");
                    }
                } else {
                    printf("Error: Invalid hysteresis format\n");
                }
            }
            // Help command
            else if (strcmp(command, "help") == 0) {
                printHelpMenu();
            }
            // Unknown command
            else {
                printf("Unknown command: '%s'\n", command);
                printf("Type 'help' for available commands\n");
            }
        }
    }

    return false;
}

bool SerialInterface::checkAlert(float temperature, float setPoint) {
    unsigned long currentTime = millis();

    // Calculate absolute deviation from setpoint
    float deviation = fabs(temperature - setPoint);

    // Check if deviation exceeds threshold and alert interval has passed
```

```
    if (deviation > ALERT_THRESHOLD) {
        if (!alertActive || (currentTime - lastAlertTime >= ALERT_INTERVAL)) {
            alertActive = true;
            lastAlertTime = currentTime;

            // Print alert message
            printf("\n!!! ALERT !!!\n");
            printf("Temperature deviation exceeds threshold: %.2f °C\n", deviation);
            printf("!!!!!!!!!!!!!!!\n");

            return true;
        }
    } else {
        // Reset alert state when temperature returns to normal range
        if (alertActive) {
            alertActive = false;
            printf("\n--- Alert cleared: Temperature stabilized ---\n");
        }
    }

    return false;
}
```

Stdio_wrapper.cpp

```
/**
 * stdio_wrapper.cpp
 * Standard I/O wrapper implementation for Arduino
 */

#include "stdio_wrapper.h"

// Create a FILE stream structure for stdin and stdout
static FILE uartout = {0};
static FILE uartin = {0};

void stdio_init(unsigned long baud) {
    // Initialize serial
    Serial.begin(baud);

    // Wait for serial port to connect (needed for native USB port only)
    while (!Serial && millis() < 5000);

    // Setup stdout stream
    fdev_setup_stream(&uartout, uart_putchar, NULL, _FDEV_SETUP_WRITE);
    stdout = &uartout;

    // Setup stdin stream
    fdev_setup_stream(&uartin, NULL, uart_getchar, _FDEV_SETUP_READ);
    stdin = &uartin;
}

int uart_putchar(char c, FILE *stream) {
    Serial.write(c);
    return 0;
```

```
}

int uart_getchar(FILE *stream) {
    // Wait until character is available
    while (!Serial.available());
    return Serial.read();
}

bool stdio_available() {
    return Serial.available() > 0;
}

bool stdio_readstring(char* buffer, size_t max_length) {
    if (!stdio_available()) {
        return false;
    }

    size_t index = 0;
    buffer[0] = '\0';

    // Wait for first character with timeout
    unsigned long start_time = millis();
    while (!stdio_available() && (millis() - start_time < 500));

    if (!stdio_available()) {
        return false;
    }

    // Read characters until newline or max length
    while (stdio_available() && index < max_length - 1) {
        char c = Serial.read();

        // Break on newline
        if (c == '\n' || c == '\r') {
            break;
        }

        // Add character to buffer
        buffer[index++] = c;
        buffer[index] = '\0';

        // Small delay to ensure all data is read
        delay(2);
    }

    // Flush any remaining characters in the buffer (up to newline)
    while (stdio_available()) {
        char c = Serial.read();
        if (c == '\n' || c == '\r') {
            break;
        }
        delay(1);
    }

    return index > 0;
```

```
}
```

Temperature_sensor.cpp

```
/**
 * temperature_sensor.cpp
 * Implementation of temperature sensor interface
 */

#include "temperature_sensor.h"
#include <DHT.h>

// Create DHT sensor instance
DHT dht(DHT_PIN, DHT_TYPE);

TemperatureSensor::TemperatureSensor() {
    lastReadTime = 0;
    currentTemperature = 0.0;
    sensorInitialized = false;
}

bool TemperatureSensor::begin() {
    dht.begin();
    delay(2000);  // DHT sensors need 2 seconds to stabilize

    // Test if sensor is responding
    float testReading = dht.readTemperature();
    if (isnan(testReading)) {
        sensorInitialized = false;
        return false;
    }

    currentTemperature = testReading;
    sensorInitialized = true;
    return true;
}

bool TemperatureSensor::update() {
    unsigned long currentTime = millis();

    // Check if it's time to read the sensor
    if (currentTime - lastReadTime >= TEMPERATURE_READ_INTERVAL) {
        lastReadTime = currentTime;

        float reading = dht.readTemperature();
        if (!isnan(reading)) {
            currentTemperature = reading;
            return true;
        }
    }

    return false;
}

float TemperatureSensor::getTemperature() {
```

```
    return currentTemperature;
}

bool TemperatureSensor::isOperational() {
    return sensorInitialized && !isnan(dht.readTemperature());
}
```