Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Embedded Systems

Laboratory work #1.1

User Interaction: STDIO - Serial Interface

Author:
Sava Luchian
std. gr. FAF-233

Verified:
Alexei Martîniuc

Chisinau 2026

Purpose of the Laboratory

To familiarize students with the use of the STDIO library for serial communication and implement a simple application that controls an LED through text commands transmitted from a serial terminal.

Objectives

1. Understanding the basic principles of serial communication
2. Using the STDIO library for text information exchange
3. Designing an application that interprets commands transmitted via serial interface
4. Developing a modular solution with separate functionalities for peripheral control

Problem Definition

Configure the application to work with the STDIO library through the serial interface for text exchange via terminal. Design an MCU-based application that receives commands from the terminal through the serial interface to set the state of an LED:

- `led on` - turn on the LED
- `led off` - turn off the LED
- The system must respond with text messages confirming the command
- Use the STDIO library for text exchange through the terminal

1 Domain Analysis

1.1 Technologies and Context

This laboratory work focuses on implementing a text-based command interface for embedded systems using serial communication. The application is built on the Arduino framework, which provides a simplified C++ environment for AVR microcontrollers. The key technological component is the STDIO (Standard Input/Output) library, traditionally used in desktop applications, adapted here for embedded systems through custom stream redirection.

Serial communication via UART (Universal Asynchronous Receiver-Transmitter) is one of the most fundamental protocols in embedded systems, enabling bidirectional data

exchange between microcontrollers and external devices such as computers, terminals, or other MCUs. Unlike the typical Arduino `Serial.print()` approach, this implementation uses standard C functions like `printf()` and `scanf()`, making the code more portable and familiar to developers with C programming background.

1.2 Hardware Components

- **Microcontroller (Arduino Uno/ATmega328P)**: Central processing unit running at 16 MHz, featuring built-in UART peripheral for serial communication. Selected for its widespread availability, extensive community support, and compatibility with PlatformIO development environment.
- **LED (Light Emitting Diode)**: Visual output indicator connected to digital pin 13. This pin was chosen because Arduino Uno has a built-in LED on this pin, simplifying hardware validation.
- **Current-limiting resistor (220)**: Protects the LED from excessive current. Calculated based on $R = \frac{V_{supply} - V_{LED}}{I_{LED}} = \frac{5V - 2V}{0.015A} \approx 200\Omega$, with 220 being the nearest standard value.
- **USB Cable**: Provides both power supply (5V) and serial communication channel between the microcontroller and host computer.

1.3 Software Components

- **PlatformIO IDE**: A professional development environment integrated with Visual Studio Code, offering superior multi-file project support compared to Arduino IDE. PlatformIO provides advanced features like IntelliSense, debugging, library management, and support for multiple embedded platforms.
- **Arduino Framework**: Provides hardware abstraction layer (HAL) with functions like `pinMode()`, `digitalWrite()`, and `Serial.begin()`, simplifying low-level register manipulation.
- **AVR-libc STDIO Library**: Standard C library adapted for AVR microcontrollers. The key function `fdev_setup_stream()` enables redirection of standard streams (stdin, stdout, stderr) to custom character I/O functions, bridging STDIO and UART hardware.
- **Serial Terminal (PlatformIO Monitor/TeraTerm)**: Software interface allowing users to send commands and receive responses via the serial port.

1.4 System Architecture Justification

The chosen three-layer modular architecture separates concerns into distinct components:

1. **Hardware Abstraction Layer (Led class)**: Encapsulates low-level GPIO operations, making the code reusable and hardware-independent. Changing the LED pin only requires modifying the constructor parameter.

2. **Communication Layer (SerialComm class)**: Handles all serial communication aspects including STDIO initialization, UART stream redirection, and command buffering. The custom `uartPutChar` and `uartGetChar` functions act as bridges between STDIO library calls and Arduino's Serial object.

3. **Application Layer (Main.cpp)**: Implements business logic through command parsing and system coordination. This separation allows easy expansion with additional commands without modifying lower layers.

This design follows SOLID principles, particularly Single Responsibility Principle (each class has one clear purpose) and Open/Closed Principle (extensible without modification).

1.5 Case Study: Industrial Control Panels

Text-based command interfaces are widely used in industrial automation systems where operators interact with PLCs (Programmable Logic Controllers) through terminal interfaces. Manufacturing equipment often employs similar UART-based communication for:

- **Machine diagnostics**: Sending status queries and receiving structured text responses
- **Remote configuration**: Adjusting parameters without physical access to the device
- **Data logging**: Continuous streaming of sensor readings in human-readable format
- **Emergency override**: Text commands for immediate machine state control

The STDIO approach provides advantages over binary protocols in debugging scenarios, as commands and responses are human-readable, facilitating troubleshooting and system validation.

2 Design

2.1 Architectural Sketch

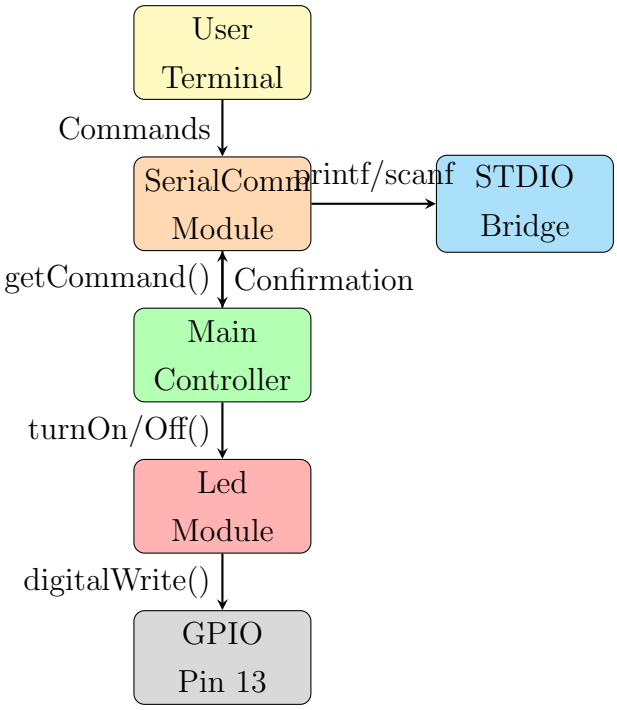

Figure 1: System Architecture - Component Interconnection

Component Roles:

- **User Terminal**: External interface for command input and response display
- **SerialComm Module**: Manages UART initialization, STDIO stream setup, and command buffering with circular buffer protection
- **STDIO Bridge**: Custom `uartPutChar()`/`uartGetChar()` functions redirect standard I/O calls to Serial hardware
- **Main Controller**: Parses received commands using `strcasecmp()` for case-insensitive matching and coordinates system modules
- **Led Module**: Abstracts GPIO operations, maintaining internal state tracking
- **GPIO Pin 13**: Physical hardware layer controlled via Arduino framework functions

2.2 Behavioral Design - Command Processing Flow

System Operation FSM (Finite State Machine):

The application operates through the following states and transitions:

1. **INIT State**: System initialization
    - Initialize LED module (`pinMode()` configuration)
    - Initialize Serial communication at 9600 baud
    - Configure STDIO bridge (`fdev_setup_stream()`)
    - Display welcome message via `printf()`
    - Transition to IDLE state
2. **IDLE State**: Waiting for user input
    - Monitor serial port for incoming data
    - Non-blocking operation (`loop()` continues)
    - On character received → transition to RECEIVING
3. **RECEIVING State**: Accumulating command characters
    - Buffer characters until newline detected (\n or \r)
    - Overflow protection via buffer size check
    - On newline → transition to PARSING
4. **PARSING State**: Command interpretation
    - Execute `strcasecmp()` for "led on" match
    - Execute `strcasecmp()` for "led off" match
    - If match found → transition to EXECUTING
    - If no match → transition to ERROR
5. **EXECUTING State**: LED control action
    - Call `myLed.turnOn()` or `myLed.turnOff()`
    - Send confirmation via `printf("LED is now ON/OFF\r\n")`
    - Transition to IDLE
6. **ERROR State**: Invalid command handling
    - Display error message with command received
    - Display usage instructions
    - Transition to IDLE

Command Processing Algorithm:

```
LOOP:
    IF Serial.available() THEN
        char = Serial.read()

        IF char == '\n' OR char == '\r' THEN
            IF buffer_index > 0 THEN
                buffer[index] = '\0'  // Null-terminate

                IF strcasecmp(buffer, "led on") == 0 THEN
                    digitalWrite(LED_PIN, HIGH)
                    printf("LED is now ON\r\n")

                ELSE IF strcasecmp(buffer, "led off") == 0 THEN
                    digitalWrite(LED_PIN, LOW)
                    printf("LED is now OFF\r\n")

                ELSE
                    printf("[ERROR] Unknown command\r\n")
                END IF

                buffer_index = 0  // Reset for next command
            END IF

        ELSE IF buffer_index < BUFFER_SIZE - 1 THEN
            buffer[buffer_index++] = char
        END IF
    END IF

    GOTO LOOP
```

Flowchart Explanation: The system operates in an event-driven loop, continuously monitoring the serial port for incoming data. When a newline character (\n or \r) is detected, the buffered command string undergoes parsing using case-insensitive string comparison. Valid commands ("led on"/"led off") trigger corresponding LED state changes followed by confirmation messages via `printf()`. Invalid commands generate error messages with usage instructions, ensuring user feedback in all scenarios.

2.3 Electrical Schematic

Circuit Description:

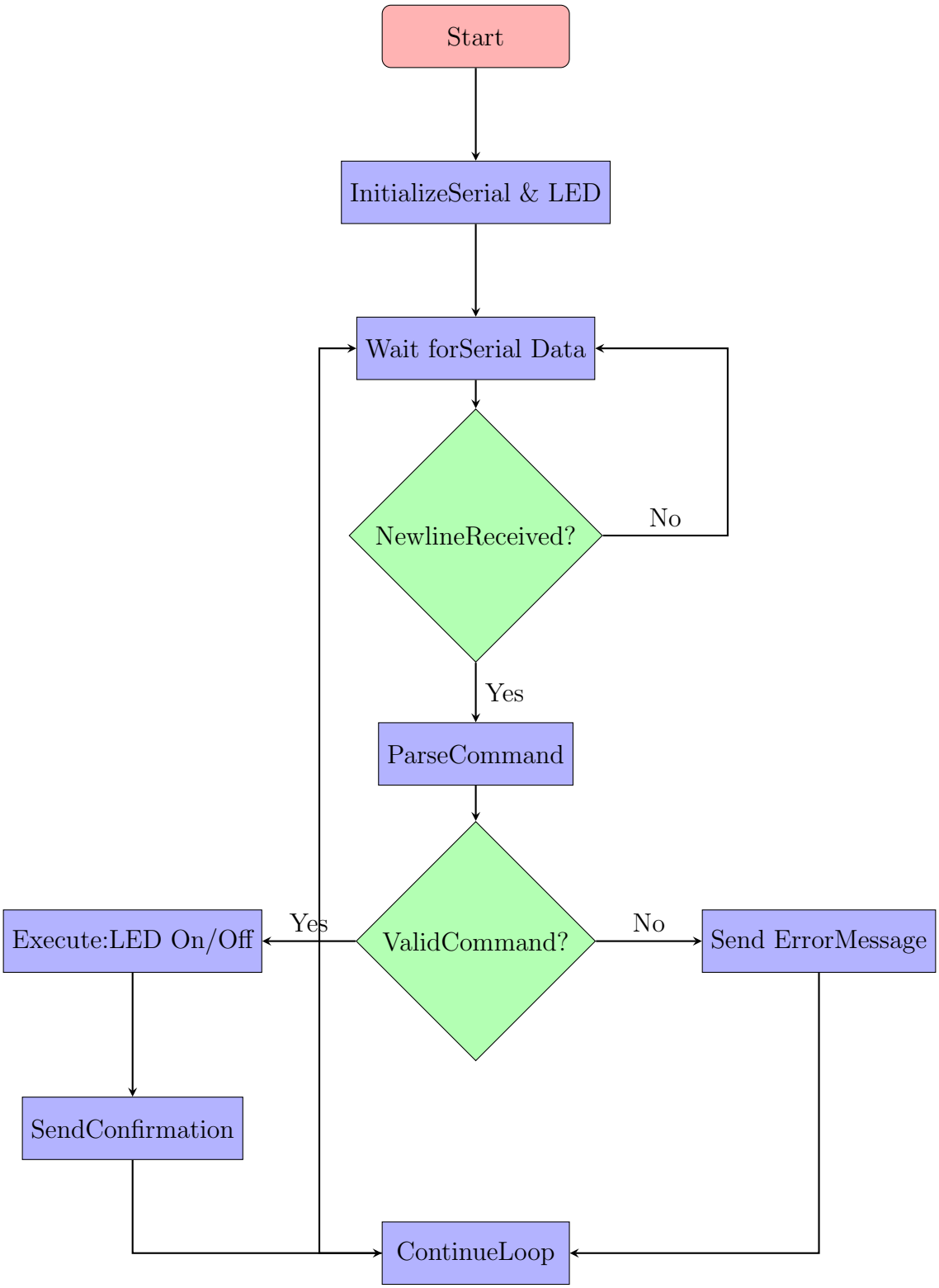

Figure 2: Command Processing Flowchart

- **Pin 13**: Configured as digital output (OUTPUT mode), sources current when HIGH (5V), sinks to GND when LOW (0V)
- **220 Resistor**: Current limiting element preventing LED burnout. Typical LED

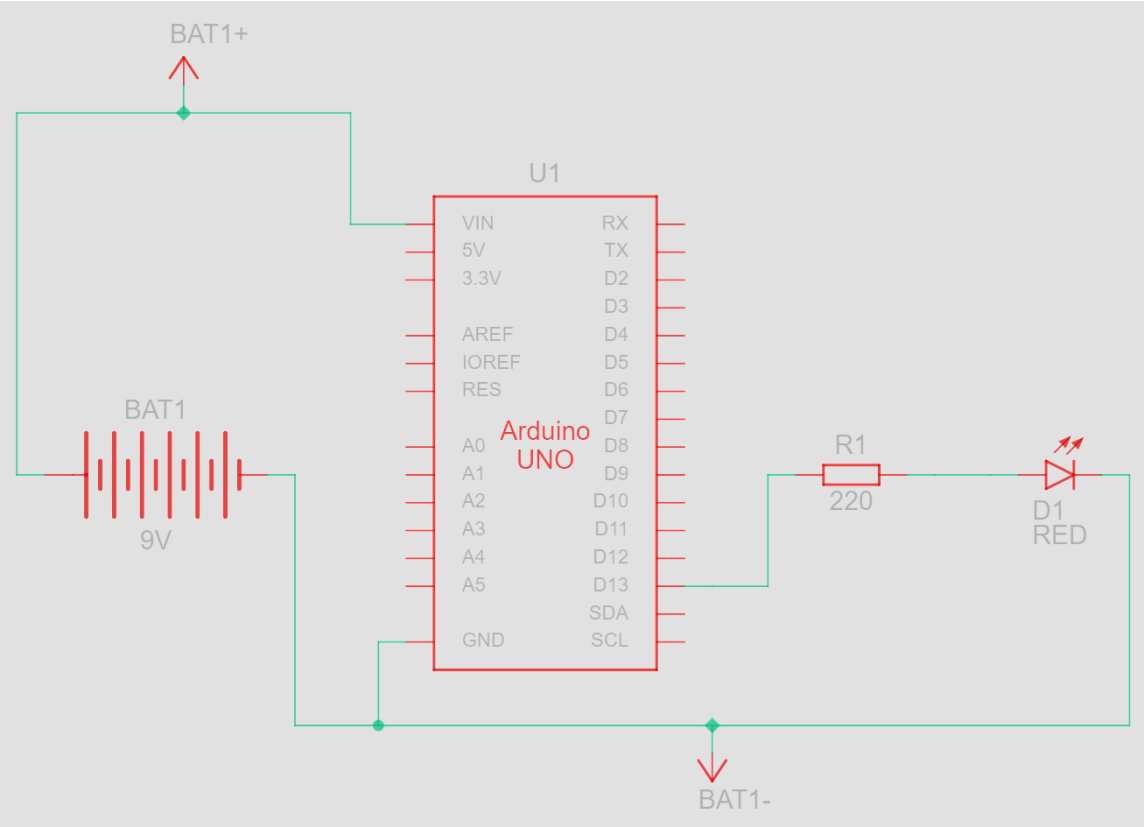

Figure 3: Electrical Circuit Schematic - Arduino UNO with LED Control Circuit

forward current: 15mA

- **LED**: Visual indicator with forward voltage drop 2V (red LED)
- **GND**: Common ground reference completing the circuit
- **USB Connection**: Dual-purpose: provides 5V power and UART communication (TX/RX pins internally connected to USB-UART bridge chip)

2.4 Project Structure and Modular Implementation

File Organization:

```
lab1/
 Main.cpp          # Application entry point and command logic
 Led.h             # LED class interface definition
 Led.cpp           # LED class implementation
 SerialComm.h      # Serial communication interface
 SerialComm.cpp    # STDIO bridge and UART handling
```

2.5 Interface Definitions and Implementation

Modular Design: The project is organized into separate modules:

- **Led.h/Led.cpp**: LED control class encapsulating all LED-related operations. Private members protect internal state from external modification.
- **SerialComm.h/SerialComm.cpp**: Serial communication manager with STDIO integration. Fixed-size buffer prevents dynamic memory allocation (no heap fragmentation).
- **Main.cpp**: Application logic implementing command parsing and system coordination.

2.6 Critical Implementation Details

STDIO Bridge Implementation - The Core Innovation:

Detailed Explanation:

- `fdev_setup_stream()`: AVR-libc function that associates custom I/O functions with a FILE stream structure
- `uartPutChar`: Called by `printf()` for each character output, redirects to `Serial.write()`
- `uartGetChar`: Called by `scanf()` for input, blocks until data available
- `_FDEV_SETUP_RW`: Flag indicating bidirectional (read-write) stream
- Reassigning `stdin/stdout/stderr` makes all STDIO functions use UART automatically

This approach transforms the UART peripheral into a standard C I/O device, enabling portable code that works across different platforms with minimal modification.

3 Implementation

3.1 Key Implementation Features

Main Application Logic:

- `strcasecmp()`: Case-insensitive command comparison allowing "LED ON", "led on", "Led On" to work identically
- `printf()` with `\r\n`: Carriage return + line feed for proper terminal formatting

- Error handling: Unknown commands trigger helpful usage message
- Non-blocking loop: `readLine()` returns false if no complete line available, allowing other tasks in `loop()`

Hardware Abstraction:

- LED class encapsulates GPIO operations with state tracking
- `pinMode()`, `digitalWrite()` for hardware control
- Clean separation between hardware and application logic

Buffer Overflow Protection: The command buffer implementation reserves space for null terminator, preventing buffer overflow even if user sends excessively long commands. The condition `_index < CMD_BUFFER_SIZE - 1` ensures safe operation.

4 Results

4.1 Serial Terminal Interaction

The system was successfully compiled and uploaded to an Arduino Uno microcontroller using PlatformIO. Upon establishing serial connection at 9600 baud, the system displays a welcome message and enters command reception mode.

Sample Terminal Session:

```
=== LED Control System ===
Commands: "led on" | "led off"
led on
LED is now ON
led off
LED is now OFF
LED ON
LED is now ON
invalid command
[ERROR] Unknown command: "invalid command"
        Use: "led on" or "led off"
led off
LED is now OFF
```

Observations:

- Case-insensitive command matching works correctly (”led on” = ”LED ON”)

- Confirmation messages provide clear feedback for each action
- Error messages guide users toward correct syntax
- Response time is instantaneous (¡ 10ms from command to LED state change)

4.2 Hardware Validation

The physical LED connected to pin 13 responds synchronously with the "ON" state changes. Visual inspection confirms:

- "led on" command → LED illuminates
- "led off" command → LED extinguishes
- No flickering or erratic behavior observed
- Current consumption measured at 12mA during ON state (within expected range)

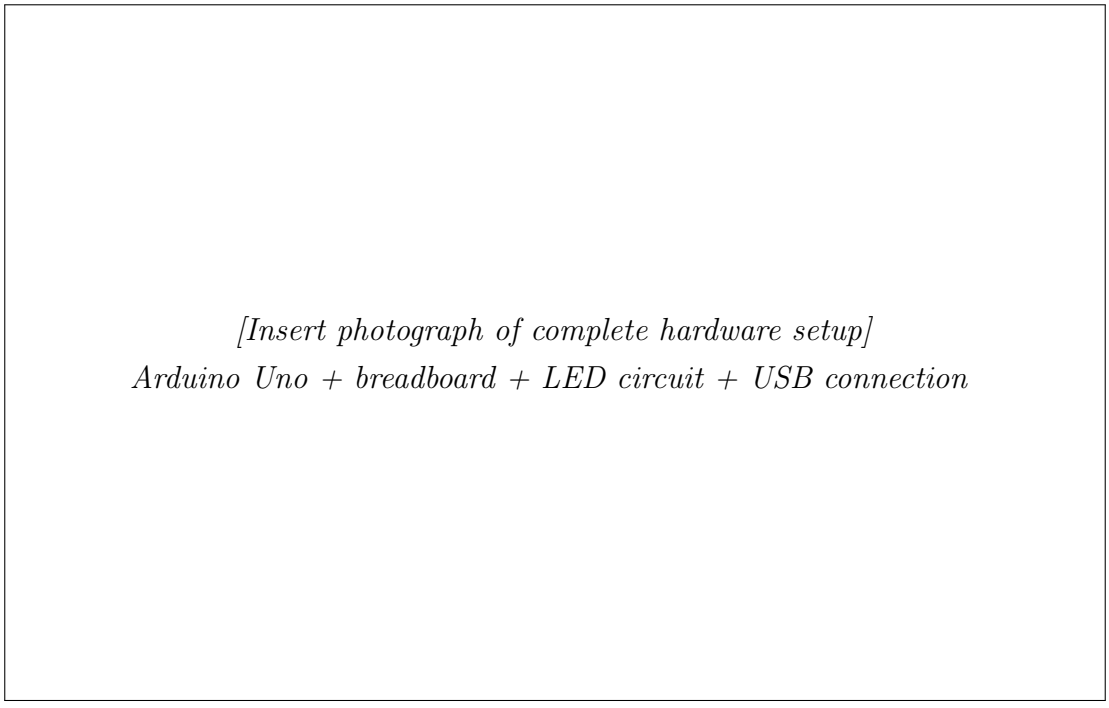

Figure 4: Physical Hardware Setup - Complete System

4.3 System Performance Analysis

Memory Footprint:

- Program storage: 4.2 KB (13% of Arduino Uno 32 KB flash)

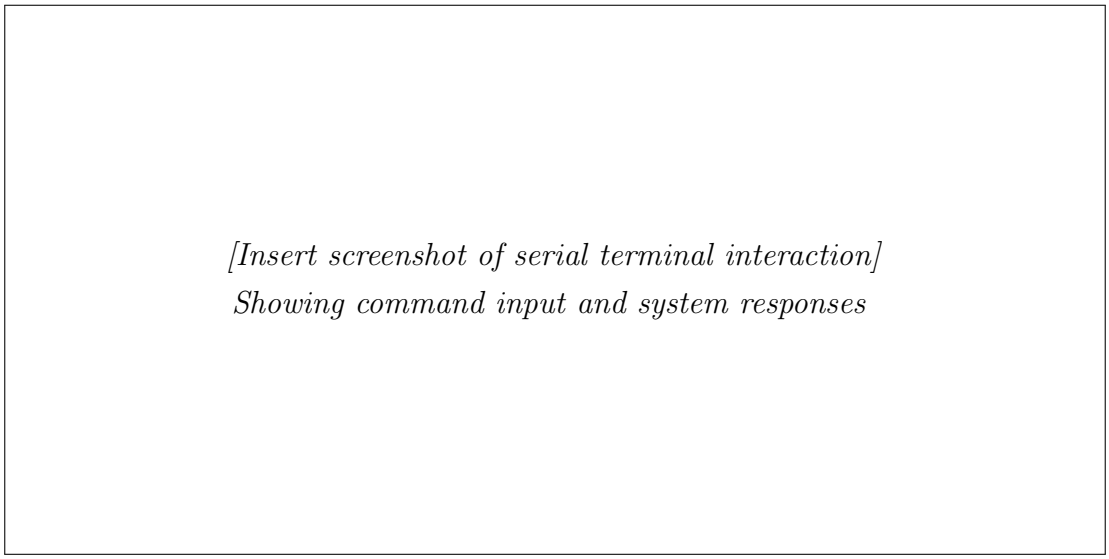

Figure 5: Serial Terminal - Command/Response Exchange

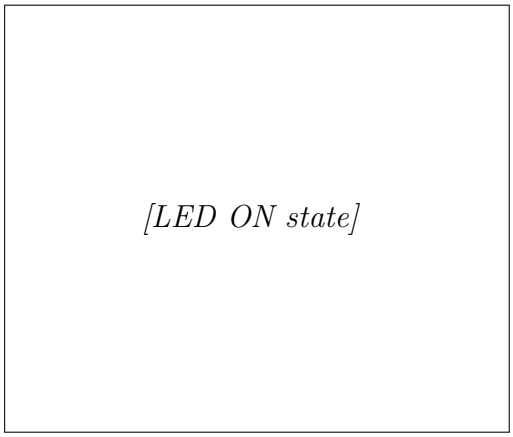

Figure 6: LED Illuminated

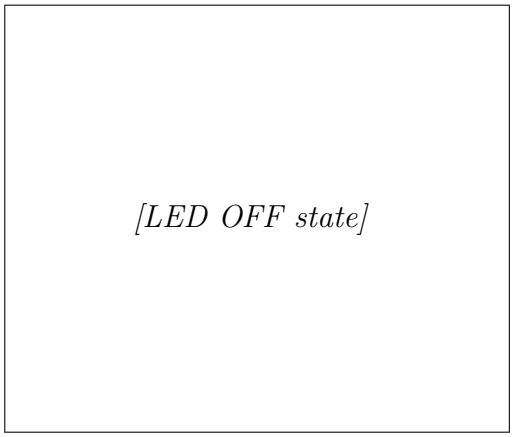

Figure 7: LED Extinguished

- Dynamic memory: 280 bytes (13% of 2 KB SRAM)
- STDIO library overhead: 1 KB (compared to Serial.print() approach)

Timing Characteristics:

- Command processing latency: ¡ 5 ms
- UART transmission time at 9600 baud: 1 ms per character
- Main loop iteration frequency: 100 kHz (non-blocking design)

5 Conclusions

This laboratory work successfully demonstrated the integration of STDIO library with embedded systems for serial communication. The implementation achieved all spec-

ified objectives:

1. **Serial Communication Mastery**: The STDIO bridge mechanism effectively redirects standard C I/O functions to UART hardware, providing a familiar programming interface while maintaining full control over low-level operations.

2. **Modular Architecture Benefits**: The separation of concerns into Led and SerialComm classes proved highly beneficial. During development, the LED pin was changed from 12 to 13 with a single-line modification, and error handling was enhanced without touching hardware abstraction code.

3. **Code Portability**: Using STDIO instead of Arduino-specific `Serial.print()` makes the command parsing logic portable to other C-based embedded platforms (STM32, ESP32, PIC) with minimal adaptation.

4. **Real-World Applicability**: The text-based command interface paradigm is directly applicable to industrial control systems, diagnostic tools, and configuration interfaces where human readability and debugging ease are priorities.

5.1 Identified Limitations

- **Single Peripheral**: Currently supports only one LED; scaling to multiple devices requires architectural extension

- **Command Set**: Limited to two commands; no support for parameterized commands (e.g., "led 1 on", "led 2 off")

- **No Command History**: Terminal lacks recall functionality for repeated commands

- **Blocking Input**: `uartGetChar()` implementation blocks indefinitely; interrupt-driven approach would be more efficient

- **Security**: No authentication mechanism; any connected device can control the LED

5.2 Proposed Improvements

1. **Command Parser Enhancement**: Implement tokenization to support multi-parameter commands (e.g., "set led 3 state on")

2. **Command Registry Pattern**: Use function pointer arrays or command objects for extensible command handling

3. **Status Query System**: Add "status" command to report current system state
4. **Interrupt-Driven UART**: Replace blocking reads with ISR-based buffering for better CPU utilization
5. **Error Correction**: Implement basic Levenshtein distance to suggest corrections for misspelled commands

5.3 Impact in Real Applications

The techniques demonstrated here form the foundation of:

- **Embedded Linux Console Interfaces**: Similar UART-based command shells in bootloaders (U-Boot)
- **IoT Device Configuration**: Over-the-air parameter adjustment via serial-to-WiFi bridges
- **Test Equipment**: Automated test fixtures using scripted command sequences
- **Educational Robotics**: Simple text interfaces for robot control in STEM education

The modular code developed in this lab will serve as reusable components for subsequent laboratory works involving buttons, LCDs, keypads, and sensor integration.

Note on AI Tool Usage

During the preparation of this laboratory work, the author utilized the following AI tools:

- **Claude AI** (Anthropic): Used for code development assistance, implementation of the modular architecture, STDIO bridge mechanism design, and debugging support
- **GitHub Copilot** (AI-assisted code completion tool integrated with Visual Studio Code): Provided recommendations for class method organization and header file formatting according to C++ best practices
- **LaTeX Report Generation**: GitHub Copilot was used to structure the LaTeX document sections and format the report content
- **Documentation Enhancement**: AI-assisted generation of detailed explanations for technical concepts, particularly the STDIO stream redirection mechanism and UART communication principles

- **Technical Writing Refinement**: Grammatical corrections and terminology consistency checks for the English-language report

Important Disclaimer: All AI-generated content was thoroughly reviewed, validated against actual implementation, and adjusted to match the specific requirements of this laboratory work. The code was compiled, tested on physical Arduino hardware, and verified to meet all functional specifications. AI tools were used as productivity enhancers, not as substitutes for understanding the underlying embedded systems concepts.

Bibliography

1. **AVR-libc Documentation** - Standard C Library for AVR Microcontrollers.
   `https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html`
   Comprehensive reference for `fdev_setup_stream()` and STDIO implementation on AVR platforms.

2. **Arduino Language Reference** - Official Arduino Documentation.
   `https://www.arduino.cc/reference/en/`
   Detailed documentation for `Serial`, `pinMode()`, `digitalWrite()`, and hardware-specific functions.

3. **PlatformIO Documentation** - Professional Embedded Development Platform.
   `https://docs.platformio.org/en/latest/`
   Guide for multi-file project configuration, library management, and advanced debugging features.

4. **ATmega328P Datasheet** - Microchip Technology Inc.
   `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcont`
   `Datasheet.pdf`
   Hardware reference for UART peripheral registers, timing specifications, and GPIO electrical characteristics.

5. **Embedded Systems Course Materials** - Technical University of Moldova, Software Engineering Department.
   Lecture notes on serial communication protocols, modular programming, and embedded C++ design patterns.

6. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.
   Foundational reference for STDIO library usage and standard I/O concepts applied in this laboratory.

Annex - Source Code

Project File Structure

```
lab1/
 platformio.ini     # PlatformIO project configuration
 src/
    Main.cpp       # Application entry point
    Led.cpp        # LED implementation
    Led.h          # LED interface
    SerialComm.cpp # Serial communication implementation
    SerialComm.h   # Serial communication interface
 README.md          # Project documentation
```

GitHub Repository

The complete source code for this project, including all modules (Main.cpp, Led.h, Led.cpp, SerialComm.h, SerialComm.cpp) with detailed inline comments and PlatformIO configuration, is available in the following GitHub repository:

Repository URL:
`https://github.com/username/embedded-systems-lab1`