Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Physics

Embedded Systems

Laboratory work #1

Development of a Dual-LED Control System with STDIO Serial Interface

Author:
Dmitrii Belih
std. gr. FAF-232

Verified:
Martiniuc A.

Chișinău 2026

1. Domain Analysis

1.1. Purpose of the Laboratory Work

The purpose of this laboratory work is to understand the basic principles of serial communication, learn to use the STDIO (Standard Input/Output) library for interaction with users, and develop a dual-LED control system on an embedded system platform. The work involves setting up the development environment, understanding the hardware-software interface, implementing a command parser for serial input, and creating a system that responds to text commands to control two LEDs independently or simultaneously. The system demonstrates the power of STDIO abstraction by using familiar C functions (`printf`, `getchar`) to handle serial communication, making embedded programming more accessible and maintainable.

1.2. Technologies Used

Standard Input/Output (STDIO)

Standard Input/Output (STDIO) is a standardized library mechanism for handling input and output operations in programming languages. In embedded systems, STDIO provides an abstraction layer that allows developers to interact with the system through formatted text input and output. The STDIO library typically includes functions such as `printf()` for formatted output and `scanf()` for formatted input. In embedded applications, STDIO is often redirected to a serial interface (UART), enabling communication between the microcontroller and a host computer via a terminal application.

Serial Communication (UART)

Universal Asynchronous Receiver-Transmitter (UART) is a hardware communication protocol used for asynchronous serial communication. UART converts parallel data from the microcontroller into serial data for transmission and vice versa. It requires only two signal lines (TX for transmission and RX for reception) plus ground, making it ideal for connecting embedded systems to computers or other devices. UART communication is characterized by configurable parameters such as baud rate (typically 9600, 115200, etc.), data bits (usually 8), parity bits, and stop bits.

GPIO (General Purpose Input/Output)

General Purpose Input/Output (GPIO) pins are configurable digital pins on microcontrollers that can be programmed as either inputs or outputs. When configured as inputs, GPIO pins can read the state of external devices such as buttons, switches, or sensors. When configured as outputs, they can drive external components such as LEDs, relays, or other digital interfaces. GPIO pins often include internal pull-up or pull-down resistors to ensure stable logic levels when external components are not actively driving the pin.

Arduino Platform

Arduino is an open-source electronics platform based on easy-to-use hardware and software. It provides a simplified development environment and a rich set of libraries that abstract away many low-level hardware details. The Arduino platform includes various microcontroller boards (such as Arduino Uno, Arduino Mega 2560, etc.), an Integrated Development Environment (IDE), and a comprehensive software framework. Arduino boards are widely used in education, prototyping, and hobby projects due to their accessibility and extensive community support.

1.3. Hardware Components

Arduino Uno

The Arduino Uno is a microcontroller board based on the ATmega328P AVR microcontroller. It features 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. The ATmega328P has 32 KB of flash memory for storing code, 2 KB of SRAM, and 1 KB of EEPROM. The board operates at 5V and can be powered via USB or an external power supply (7-12V).

LED (Light Emitting Diode)

A Light Emitting Diode (LED) is a semiconductor light source that emits light when current flows through it. LEDs are polarized components, meaning they must be connected in the correct orientation. Typically, an LED requires a forward voltage drop of approximately 1.8V to 3.3V (depending on color) and a forward current of 20 mA for optimal brightness. To limit the current and prevent damage to both the LED and the microcontroller, a current-limiting resistor must be used in series with the LED. For a 5V supply and a typical LED with 2V forward voltage, a 150-220 ohm resistor is appropriate.

1.4. Software Components

Arduino IDE

The Arduino Integrated Development Environment (IDE) is a cross-platform application written in Java that provides a simplified programming environment for Arduino boards. It includes a code editor with syntax highlighting, a compiler based on GCC, a library manager, and tools for uploading code to the microcontroller. The Arduino IDE supports

C++ programming with a simplified API that includes functions such as `digitalWrite()`, `digitalRead()`, `analogRead()`, and `Serial` methods for serial communication.

PlatformIO

PlatformIO is an open-source ecosystem for embedded development that integrates with various code editors (including VS Code). It provides advanced features such as intelligent code completion, multi-platform build systems, library management, unit testing, and debugging capabilities. PlatformIO supports a wide range of development boards and frameworks, making it a powerful alternative to the Arduino IDE for more complex projects.

1.5. System Architecture and Justification

The system architecture follows a layered approach that separates hardware abstraction, driver implementation, and application logic:

- **Hardware Layer:** Consists of the Arduino Uno microcontroller, two LEDs, and UART serial interface. The microcontroller provides GPIO pins and UART hardware for communication.
- **Hardware-Software Interface (Driver Layer):** This layer implements drivers for hardware components. The LED driver provides functions to turn LEDs on and off, while the serial driver abstracts the UART hardware and provides STDIO functions.
- **Application Layer:** Implements the main application logic, responding to serial commands and controlling the LEDs accordingly. The application uses STDIO for user communication, providing status messages and receiving commands.
- **STDIO Service Layer:** Provides formatted input/output capabilities by redirecting STDIO functions to the serial interface, enabling communication with a terminal application on a host computer.

This architecture was chosen because it promotes modularity, reusability, and separation of concerns. Each layer has a well-defined responsibility, making the code easier to understand, maintain, and extend. The use of STDIO for serial communication simplifies user interaction and leverages familiar I/O functions from standard programming.

1.6. Case Study: Interactive Embedded Systems

Interactive embedded systems are widely used in consumer electronics, industrial control, and IoT applications. For example, a smart home lighting system typically allows users to control multiple lights through a mobile app or voice commands. The system receives text or voice commands, parses them, and sends control signals to individual lights. This is analogous to our dual-LED control system, where text commands via serial interface control individual LEDs or groups of LEDs.

Another example is an industrial machine control interface that accepts commands through a terminal or HMI (Human-Machine Interface). Operators type commands to start, stop, or configure machinery, and the system provides status feedback through LEDs and text responses. The command parsing and LED control techniques implemented in this laboratory work are fundamental building blocks for such systems.

The STDIO approach used here is particularly valuable because it allows embedded systems to use familiar I/O functions (`printf`, `scanf`, `getchar`) that abstract away the details of serial communication. This abstraction layer makes embedded code more readable, portable, and easier to develop for programmers who are experienced with standard C programming.

2. Design

2.1. Architectural Sketch

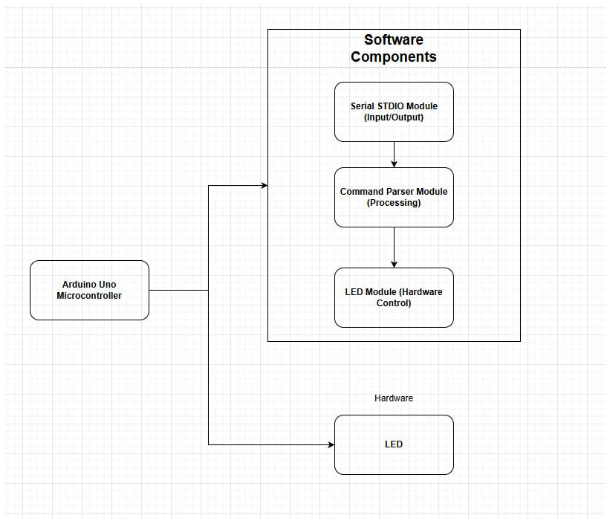

Software Components
Serial STDIO Module (Input/Output)
Command Parser Module (Processing)
LED Module (Hardware Control)
Arduino Uno Microcontroller
Hardware
LED

Рис. 1: System Architecture

Architecture Components:

- **User (Serial Terminal Input):** User interacts with the system through a serial terminal, entering text commands to control the LEDs.
- **Serial STDIO Module (Input/Output):** Handles communication between the microcontroller and the terminal, redirecting standard C I/O functions (`printf`, `getchar`) to the serial interface.

- **Command Parser Module (Processing):** Interprets user commands, performing text processing (trimming whitespace, case conversion) and matching against known command patterns.
- **LED Module (Hardware Control):** Controls the physical LEDs based on parsed commands, managing GPIO pins and providing visual feedback.

2.3. Functional Block Diagrams

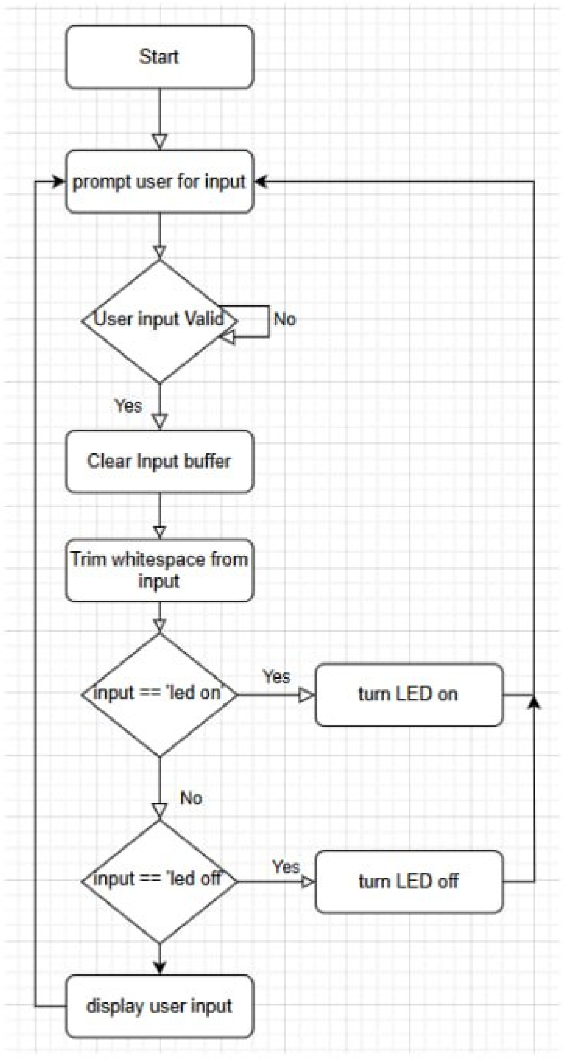


Рис. 2: Application Logic Flowchart

Application Algorithm:

1. **Start:** Begin program execution.
2. **Prompt User for Input:** Display prompt and wait for user input via serial interface.

3. **User Input Valid?:** Check if input is valid (not empty, proper format).
   - **No:** Return to prompt user for input.
   - **Yes:** Proceed to next step.
4. **Clear Input Buffer:** Clear any remaining characters in the input buffer.
5. **Trim Whitespace from Input:** Remove leading and trailing whitespace characters.
6. **input == 'led on'?:** Check if input matches "led on"command.
   - **Yes:** Turn LED on, then return to prompt user for input.
   - **No:** Proceed to next check.
7. **input == 'led off'?:** Check if input matches "led off"command.
   - **Yes:** Turn LED off, then return to prompt user for input.
   - **No:** Display user input (unknown command), then return to prompt user for input.

The system uses a continuous loop that prompts for input, validates it, processes it through command matching, and provides appropriate feedback. This interactive approach allows for real-time LED control through a simple command-line interface.

2.4. Electrical Schematic

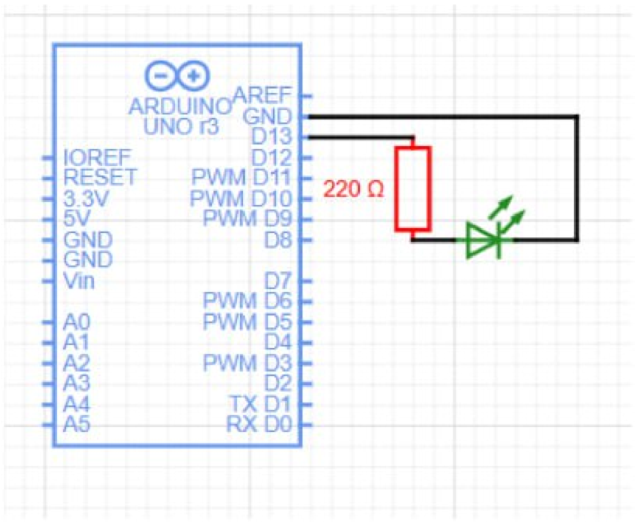


Рис. 3: Electrical Schematic

Circuit Description:

LED1 Circuit:

- **Connection:** LED1 anode (positive terminal) is connected to Pin 9 of the Arduino through a 220 ohm current-limiting resistor. LED1 cathode (negative terminal) is connected to ground (GND).
- **Current Calculation:** With a 5V supply, 2V forward voltage drop across the LED, and 220 ohm resistor, the current is: $I = (5V - 2V)/220\Omega = 13.6mA$, which is within the safe operating range for both the LED and the Arduino GPIO pin (maximum 40 mA per pin).
- **Operation:** When Pin 9 is set HIGH (5V), current flows from the pin through the resistor and LED1 to ground, causing the LED to light up. When Pin 9 is set LOW (0V), no current flows and the LED is off.

LED2 Circuit:

- **Connection:** LED2 anode (positive terminal) is connected to Pin 13 of the Arduino through a 220 ohm current-limiting resistor. LED2 cathode (negative terminal) is connected to ground (GND).
- **Current Calculation:** Same as LED1: $I = (5V - 2V)/220\Omega = 13.6mA$.
- **Operation:** When Pin 13 is set HIGH (5V), LED2 lights up. When Pin 13 is set LOW (0V), LED2 is off. Note that Pin 13 also has an onboard LED connected to it, so LED2 status can be observed both on the breadboard and the Arduino board.

2.5. Project Structure

The project follows a modular structure where each hardware component is represented by separate header (.h) and implementation (.cpp) files:

```
ES/
|-- src/
|   |-- main.cpp                          # Main application logic
|   ‘-- modules/
|       |-- led/
|       |   |-- led.h                     # LED driver interface
|       |   ‘-- led.cpp                   # LED driver implementation
|       |-- command/
|       |   |-- command.h                 # Command parser interface
|       |   ‘-- command.cpp               # Command parser implementation
|       ‘-- serial_stdio/
|           |-- serial_stdio.h            # STDIO serial interface
|           `-- serial_stdio.cpp          # STDIO implementation
|-- platformio.ini                        # PlatformIO configuration
`-- lib/                                  # Additional libraries (if any)
```

Module Descriptions:

LED Module (led.h / led.cpp)

Interface (led.h):

- `Led::Led(uint8_t pin)`: Constructor that initializes the LED with the specified GPIO pin.
- `void Led::begin()`: Configures the GPIO pin as output and initializes LED state to off.
- `void Led::on()`: Turns the LED on by setting the GPIO pin to HIGH.
- `void Led::off()`: Turns the LED off by setting the GPIO pin to LOW.
- `void Led::toggle()`: Toggles the LED state (on to off, off to on).
- `bool Led::state() const`: Returns the current LED state (true = on, false = off).

Implementation (led.cpp): The LED driver uses Arduino's `pinMode()` function to configure the GPIO pin as output and `digitalWrite()` to control the pin state. The driver maintains an internal state variable to track the current LED status.

Command Module (command.h / command.cpp)

Interface (command.h):

- `enum CommandType`: Enumeration of supported commands (LED1_ON, LED1_OFF, LED2_ON, LED2_OFF, BOTH_ON, BOTH_OFF, UNKNOWN, EMPTY).
- `static CommandType CommandParser::parse(const char* input)`: Parses a text string and returns the corresponding command type.
- `static bool CommandParser::isValid(CommandType cmd)`: Checks if a command type is valid.
- `static const char* CommandParser::toString(CommandType cmd)`: Converts a command type to its string representation.

Implementation (command.cpp): The command parser implements text processing functions including whitespace trimming and case conversion. It compares the processed input string against known command patterns and returns the appropriate command type. The parser is case-insensitive and handles leading/trailing whitespace gracefully.

Serial STDIO Module (serial_stdio.h / serial_stdio.cpp)

Interface (serial_stdio.h):

- `static void SerialStdio::begin(unsigned long baudRate)`: Initializes serial communication and redirects stdout/stdin to the serial port.
- `static void SerialStdio::printWelcome()`: Prints the welcome message and available commands.
- `static int SerialStdio::readLine(char* buffer, int bufferSize)`: Reads a line of text from serial input until newline character.

Implementation (serial_stdio.cpp): The STDIO module uses AVR-libc's file stream functions (`fdev_setup_stream`) to redirect standard C I/O functions (`printf`, `getchar`) to the Arduino's Serial port. This allows the use of familiar C library functions for serial communication. The module handles character echoing for backspace and provides line buffering.

Main Application (main.cpp)

Setup:

- Initialize serial communication at 9600 baud with STDIO redirection.
- Initialize LED1 on Pin 9 and LED2 on Pin 13.
- Print welcome message with available commands.

Main Loop:

- Read a line of text input from serial interface.
- Parse the input string to determine the command type.
- Print debug information showing the received command.
- Execute the command by controlling the appropriate LEDs.
- Print confirmation message indicating the result.
- Repeat indefinitely.

The application demonstrates the integration of command parsing, LED control, and STDIO for user interaction, following the layered architecture design.

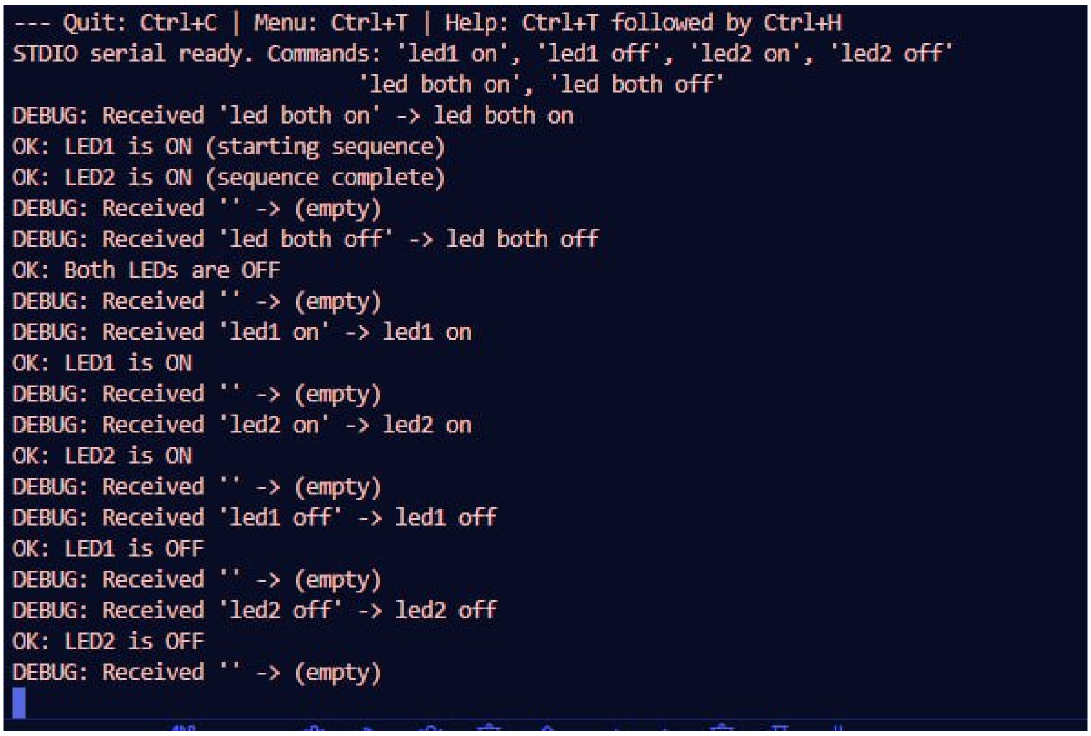
3. Results

3.1. System Operation

The dual-LED control system was successfully implemented and tested. The system responds to text commands via the serial interface, controlling two LEDs independently or simultaneously. The STDIO abstraction allows the use of familiar C functions (`printf`, `getchar`) for serial communication, demonstrating the power and simplicity of this approach.

3.2. Serial Interface Output

The following output was captured from the serial interface during system operation:



```
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
STDIO serial ready. Commands: 'led1 on', 'led1 off', 'led2 on', 'led2 off'
                           'led both on', 'led both off'
DEBUG: Received 'led both on' -> led both on
OK: LED1 is ON (starting sequence)
OK: LED2 is ON (sequence complete)
DEBUG: Received '' -> (empty)
DEBUG: Received 'led both off' -> led both off
OK: Both LEDs are OFF
DEBUG: Received '' -> (empty)
DEBUG: Received 'led1 on' -> led1 on
OK: LED1 is ON
DEBUG: Received '' -> (empty)
DEBUG: Received 'led2 on' -> led2 on
OK: LED2 is ON
DEBUG: Received '' -> (empty)
DEBUG: Received 'led1 off' -> led1 off
OK: LED1 is OFF
DEBUG: Received '' -> (empty)
DEBUG: Received 'led2 off' -> led2 off
OK: LED2 is OFF
DEBUG: Received '' -> (empty)
```

Рис. 4: Serial Interface Console Output

Output Analysis:

The system initializes successfully with welcome message and available commands. Commands are parsed and executed with debug feedback showing received input and confirmation messages. The "led both on"command demonstrates sequential activation with 500ms delay between LEDs. Empty input is handled gracefully.

3.3. System Screenshots

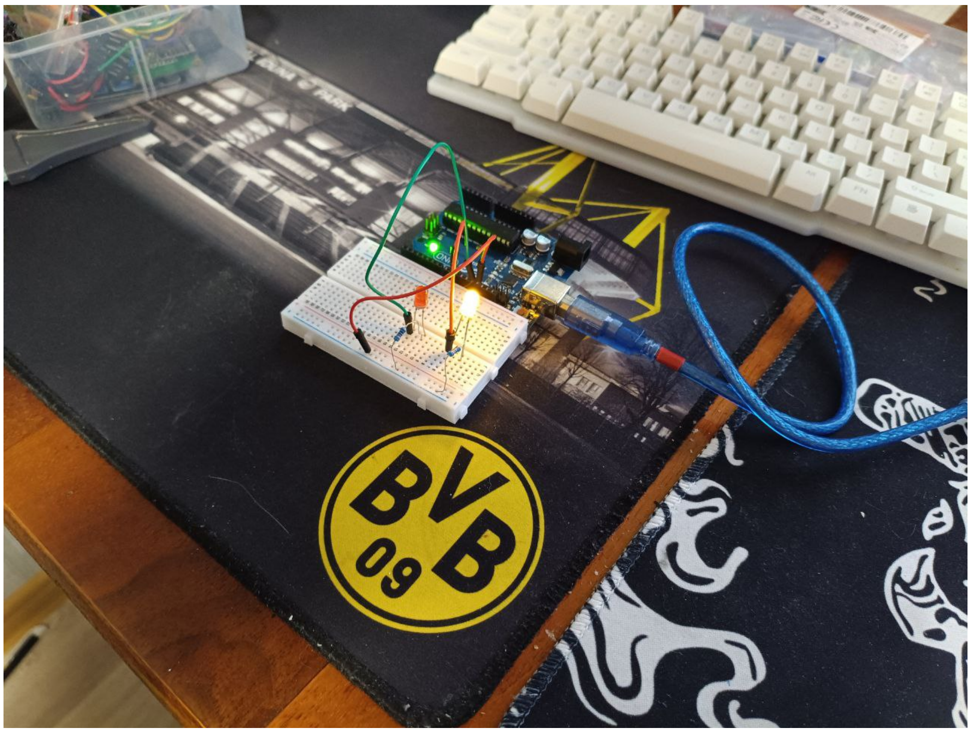


Рис. 5: System Running with Both LEDs On

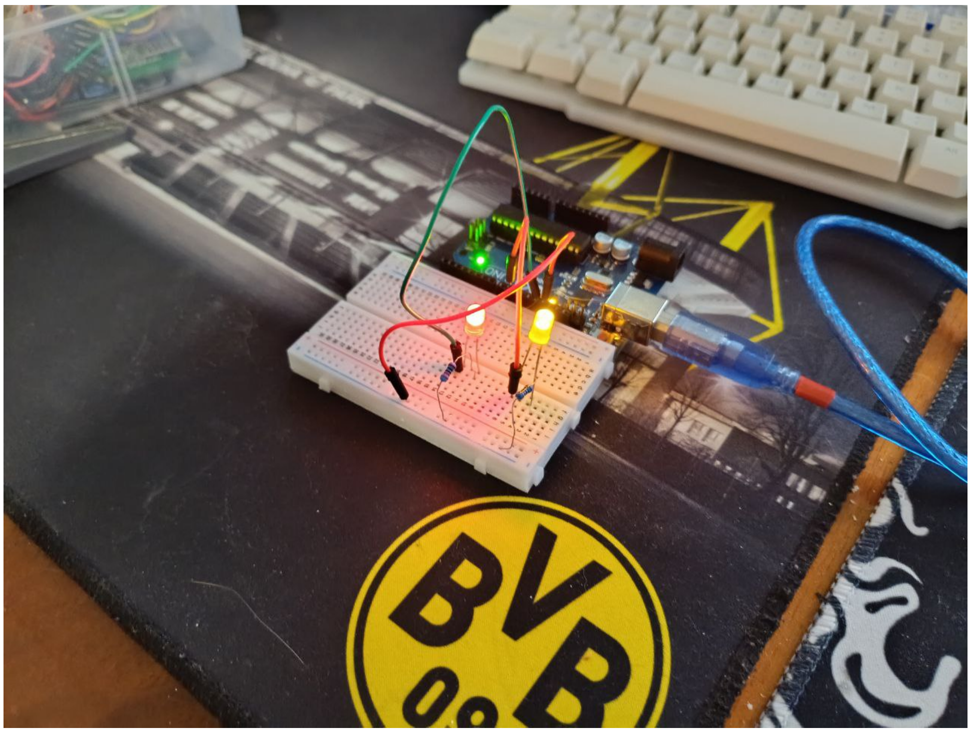


Рис. 6: System Running with Both LEDs On (Alternate View)

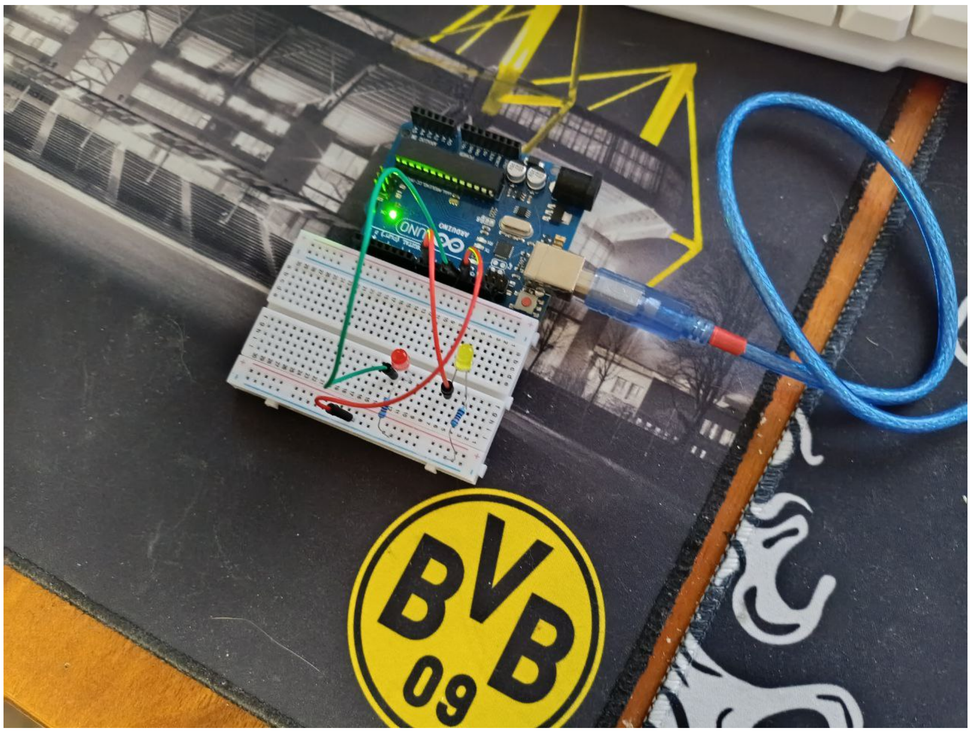


Рис. 7: System Running with Both LEDs Off

Screenshot Analysis:

Figures ?? and ?? show both LEDs illuminated (yellow LED1 and red LED2) after executing "led both on"command. Figure ?? shows both LEDs off. The Arduino board is powered and operational, with LEDs connected through 220Ω resistors on a breadboard.

3.4. Hardware Montage

The hardware montage shows:

- Arduino Uno development board (or compatible)
- Two LEDs connected to Pins 9 (LED1 - yellow/orange) and 13 (LED2 - red)
- Two 220 ohm current-limiting resistors
- Breadboard for prototyping connections
- Multicolored jumper wires for connections
- USB cable for power and serial communication to the host computer

The setup demonstrates a clean, modular approach to embedded systems prototyping, with clear separation between the microcontroller, passive components (LEDs, resistors), and interconnections.

4. Conclusions

4.1. Performance Analysis

The dual-LED control system demonstrated reliable operation. Command response is immediate with efficient parsing (case-insensitive, whitespace handling). Serial communication at 9600 baud proved stable with successful STDIO redirection. LED control is reliable with no flickering. Memory usage is minimal (4KB flash, 300 bytes RAM). Modular design with separate modules for LED, command parsing, and serial STDIO ensures maintainability.

4.2. Limitations and Identified Issues

- **Blocking I/O:** System waits for user input, preventing concurrent tasks or background processing.
- **Limited Command Set:** Only six basic commands; no status queries, PWM brightness control, or sequences.
- **No Error Recovery:** Minimal error handling for invalid commands or hardware failures.
- **No State Persistence:** LED states not saved across power cycles.
- **Fixed Baud Rate:** Hardcoded to 9600 baud.
- **No Authentication:** Accepts commands from any source without access control.

4.4. Knowledge Gained

Gained practical experience with PlatformIO environment, embedded system architecture, GPIO control, serial communication with STDIO redirection, command parsing with text processing, modular code design, debugging via serial output, AVR-libc file stream functions, and abstraction layer benefits.

4.5. Real-World Applications

Techniques apply to smart home lighting systems, industrial machine controllers, network equipment configuration consoles, automotive diagnostic tools, medical equipment interfaces, and IoT gateways. The foundational concepts—GPIO control, serial communication, command parsing, STDIO abstraction, and modular design—are essential for embedded systems engineering.

5. Note on AI Tools Usage

During the preparation of this report, the author utilized ChatGPT (an AI language model developed by OpenAI) for generating and consolidating content. The AI assistance was used for:

- Generating and structuring technical descriptions of hardware components and technologies.
- Formulating explanations of system architecture and design decisions.
- Drafting sections on domain analysis and case studies.
- Suggesting improvements and limitations based on the implemented solution.
- Assisting with formatting and organizing the report structure.

All information generated by the AI tool was reviewed, validated, and adjusted by the author to ensure accuracy, relevance, and compliance with the laboratory work requirements. The author takes full responsibility for the content presented in this report.

6. Bibliography

1. Arduino.cc. *Arduino Language Reference.* Available: `https://www.arduino.cc/reference/en/` [Accessed: 2026-02-08].
2. Atmel Corporation. *ATmega328P Datasheet - Complete.* 2014. Available: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-A Datasheet.pdf` [Accessed: 2026-02-08].
3. PlatformIO. *PlatformIO Documentation.* Available: `https://docs.platformio.org/` [Accessed: 2026-02-08].
4. Axelson, J. *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems.* Lakeview Research, 2007.
5. Arduino.cc. *Serial Communication.* Available: `https://www.arduino.cc/reference/en/language/functions/communication/serial/` [Accessed: 2026-02-08].

7. Appendix - Source Code

7.1. LED Driver Header (led.h)

Листинг 1: led.h - LED Driver Interface

```cpp
#ifndef LED_H
#define LED_H

#include <Arduino.h>

class Led
{
public:
\texplicit Led(uint8_t pin);
  void begin();
  void on();
  void off();
  void toggle();
  bool state() const;

private:
  uint8_t _pin;
  bool _state;
};

#endif // LED_H
```

7.2. LED Driver Implementation (led.cpp)

Листинг 2: led.cpp - LED Driver Implementation

```cpp
#include "led.h"

Led::Led(uint8_t pin)
    : _pin(pin), _state(false) {}

void Led::begin()
{
  pinMode(_pin, OUTPUT);
  digitalWrite(_pin, LOW);
  _state = false;
}

void Led::on()
{
  digitalWrite(_pin, HIGH);
  _state = true;
}

void Led::off()
```

```cpp
{
  digitalWrite(_pin, LOW);
  _state = false;
}

void Led::toggle()
{
  if (_state)
    off();
  else
    on();
}

bool Led::state() const { return _state; }
```

7.3. Command Parser Header (command.h)

Листинг 3: command.h - Command Parser Interface

```cpp
#ifndef COMMAND_H
#define COMMAND_H

#include <Arduino.h>

enum CommandType {
    CMD_LED1_ON,
    CMD_LED1_OFF,
    CMD_LED2_ON,
    CMD_LED2_OFF,
    CMD_BOTH_ON,
    CMD_BOTH_OFF,
    CMD_UNKNOWN,
    CMD_EMPTY
};

class CommandParser {
public:
    static CommandType parse(const char* input);

    static bool isValid(CommandType cmd);

    static const char* toString(CommandType cmd);

private:
    static const int MAX_CMD_LENGTH = 32;
};
```

```

#endif // COMMAND_H
```

7.4. Command Parser Implementation (command.cpp)

Листинг 4: command.cpp - Command Parser Implementation

```
#include "command.h"
#include <string.h>
#include <ctype.h>

static void trimWhitespace(char* str) {
    char* end;

    while (isspace((unsigned char)*str)) str++;

    if (*str == 0) {
        *str = 0;
        return;
    }

    end = str + strlen(str) - 1;
    while (end > str && isspace((unsigned char)*end)) end--;

    *(end + 1) = 0;
}

static void toLowerCase(char* str) {
    for (; *str; ++str) {
        *str = tolower((unsigned char)*str);
    }
}

CommandType CommandParser::parse(const char* input) {
    if (input == nullptr || input[0] == '\0') {
        return CMD_EMPTY;
    }

    char buffer[MAX_CMD_LENGTH];
    strncpy(buffer, input, MAX_CMD_LENGTH - 1);
    buffer[MAX_CMD_LENGTH - 1] = '\0';

    trimWhitespace(buffer);
    toLowerCase(buffer);

    if (strcmp(buffer, "led1 on") == 0) {
        return CMD_LED1_ON;
    } else if (strcmp(buffer, "led1 off") == 0) {
        return CMD_LED1_OFF;
    } else if (strcmp(buffer, "led2 on") == 0) {
        return CMD_LED2_ON;
    } else if (strcmp(buffer, "led2 off") == 0) {
        return CMD_LED2_OFF;
    } else if (strcmp(buffer, "led both on") == 0) {
        return CMD_BOTH_ON;
    } else if (strcmp(buffer, "led both off") == 0) {
        return CMD_BOTH_OFF;
    }

    return CMD_UNKNOWN;
}

bool CommandParser::isValid(CommandType cmd) {
    return (cmd >= CMD_LED1_ON && cmd <= CMD_BOTH_OFF) || cmd == CMD_EMPTY;
}

const char* CommandParser::toString(CommandType cmd) {
    switch (cmd) {
        case CMD_LED1_ON:      return "led1 on";
        case CMD_LED1_OFF:     return "led1 off";
        case CMD_LED2_ON:      return "led2 on";
        case CMD_LED2_OFF:     return "led2 off";
        case CMD_BOTH_ON:      return "led both on";
        case CMD_BOTH_OFF:     return "led both off";
        case CMD_EMPTY:        return "(empty)";
        case CMD_UNKNOWN:
        default:               return "unknown";
    }
}
```

7.5. Serial STDIO Header (serial_stdio.h)

Листинг 5: serial$_{s}tdio.h - SerialSTDIOInterface$

```
#ifndef SERIAL_STDIO_H
#define SERIAL_STDIO_H

#include <Arduino.h>

class SerialStdio {
public:
    static void begin(unsigned long baudRate);

    static void printWelcome();

    static int readLine(char* buffer, int bufferSize);

public:
    static const int LINE_BUF_SIZE = 80;

private:
    static int serialPutchar(char c, FILE* stream);

    static int serialGetchar(FILE* stream);

    static FILE serial_stdout;
    static FILE serial_stdin;
};

#endif // SERIAL_STDIO_H
```

7.6. Serial STDIO Implementation (serial_stdio.cpp)

Листинг 6: serial$_s tdio.cpp - SerialSTDIOImplementation$

```
#include "serial_stdio.h"
#include <stdio.h>

FILE SerialStdio::serial_stdout;
FILE SerialStdio::serial_stdin;

void SerialStdio::begin(unsigned long baudRate) {
    Serial.begin(baudRate);

    fdev_setup_stream(&serial_stdout, serialPutchar, NULL,
   _FDEV_SETUP_WRITE);
    fdev_setup_stream(&serial_stdin, NULL, serialGetchar,
   _FDEV_SETUP_READ);
    stdout = &serial_stdout;
    stdin = &serial_stdin;
}

void SerialStdio::printWelcome() {
    printf("STDIO serial ready. Commands: 'led1 on', 'led1 off', 'led2
   on', 'led2 off'\n");
    printf("                           'led both on', 'led both off'\n"
   );
}

int SerialStdio::readLine(char* buffer, int bufferSize) {
    int pos = 0;

    while (true) {
        int c = getchar();

        if (c == '\r' || c == '\n') {
            buffer[pos] = '\0';
            return pos;
        }

        if (c == '\b' || c == 127) {
            if (pos > 0) {
                pos--;
            }
            continue;
        }

        if (c >= 32 && c <= 126 && pos < bufferSize - 1) {
            buffer[pos++] = (char)c;
        }
    }
}

int SerialStdio::serialPutchar(char c, FILE* stream) {
    if (c == '\n') Serial.write('\r');
    Serial.write(c);
    return 0;
}

int SerialStdio::serialGetchar(FILE* stream) {
    while (!Serial.available());
    int c = Serial.read();

    if (c == '\b' || c == 127) {
        Serial.write("\b \b");
        return c;
    }

    return c;
}
```

7.7. Main Application (main.cpp)

Листинг 7: main.cpp - Main Application

```
#include <Arduino.h>
#include "serial_stdio/serial_stdio.h"
#include "command/command.h"
#include "led/led.h"

static const unsigned long SERIAL_BAUD_RATE = 9600;
static const uint8_t LED1_PIN = 9;
static const uint8_t LED2_PIN = 13;

static Led led1(LED1_PIN);
static Led led2(LED2_PIN);

static void executeCommand(CommandType cmd)
{
  switch (cmd)
  {
  case CMD_LED1_ON:
    led1.on();
    printf("OK: LED1 is ON\n");
    break;

  case CMD_LED1_OFF:
    led1.off();
    printf("OK: LED1 is OFF\n");
    break;

  case CMD_LED2_ON:
    led2.on();
    printf("OK: LED2 is ON\n");
    break;

  case CMD_LED2_OFF:
    led2.off();
    printf("OK: LED2 is OFF\n");
    break;

  case CMD_BOTH_ON:
    led1.on();
    printf("OK: LED1 is ON (starting sequence)\n");
    delay(500);
    led2.on();
    printf("OK: LED2 is ON (sequence complete)\n");
    break;

  case CMD_BOTH_OFF:
    led1.off();
    led2.off();
    printf("OK: Both LEDs are OFF\n");
    break;

  case CMD_EMPTY:
    break;

  case CMD_UNKNOWN:
  default:
    printf("ERR: Unknown command\n");
    break;
  }
}

void setup()
{
  SerialStdio::begin(SERIAL_BAUD_RATE);

  led1.begin();
  led2.begin();

  SerialStdio::printWelcome();
}

void loop()
{
  char line[SerialStdio::LINE_BUF_SIZE];

  SerialStdio::readLine(line, sizeof(line));

  CommandType cmd = CommandParser::parse(line);

  printf("DEBUG: Received '%s' -> %s\n",
         line,
         CommandParser::toString(cmd));

  executeCommand(cmd);
}
```