MINISTRY OF EDUCATION AND RESEARCH
OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software and Automation Engineering

VREMERE ADRIAN, FAF-232

Report

Laboratory Work n.1.1

User Interaction: STDIO — Serial Interface

on Embedded Systems

Checked by:
Martîniuc Alexei, *univ. asist.*
FCIM, UTM

Chişinău – 2026

1. Domain Analysis

1.1. Objective of the Laboratory Work

The objective of this laboratory work is to familiarize students with the use of the C Standard I/O (STDIO) library for serial communication in embedded systems. The primary goal is to design and implement a simple application on an Arduino Mega 2560 microcontroller that controls an LED through text commands transmitted from a serial terminal.

The key learning objectives include understanding the fundamental principles of serial communication (UART protocol), using the STDIO library functions (`printf`, `fgets`) for text-based data exchange over a serial interface, designing an application that interprets commands received through the serial port, and developing a modular software solution with separate reusable components for each peripheral device.

1.2. Problem Definition

The task requires the development of a microcontroller-based application with the following functional requirements:

1. Configure the application to use the STDIO library through the serial interface for text exchange via a terminal emulator.
2. Design and implement a command interpreter that receives commands from the serial terminal:
   - `led on` — turns the LED ON.
   - `led off` — turns the LED OFF.
3. The system must respond with text confirmation messages for each processed command.
4. All text input/output must be handled using the C STDIO library (`printf`, `fgets`, etc.), not Arduino-specific `Serial.print()` calls at the application level.
5. The application must be structured in a modular fashion, with separate files for each peripheral (LED, serial I/O).

1.3. Used Technologies

UART Serial Communication

UART (Universal Asynchronous Receiver-Transmitter) is a hardware communication protocol that enables asynchronous serial data exchange between two devices. In this laboratory, UART is used to establish communication between the Arduino Mega 2560 and a host PC running a serial terminal application.

UART communication is asynchronous, meaning there is no shared clock signal between transmitter and receiver; both sides must agree on a common baud rate (9600 in this lab). It operates in full-duplex mode, allowing data to be transmitted and received simultaneously over two separate lines (TX and RX). The frame format consists of a start bit, 8 data bits, an optional parity bit, and one or more stop bits. UART is also a point-to-point protocol, connecting exactly two devices directly.

Asynchronous data transmission occurs at the frame level, each having a specific structure as shown in *Figure 1.1*. At the protocol level, UART defines several key communication parameters: baud rate (transfer speed), data bits (typically 8 bits per frame), parity (for error detection), and stop bits (to mark frame boundaries). A commonly encountered standard configuration is 9600 baud, 8 data bits, no parity, and 1 stop bit (abbreviated as 9600 8N1).

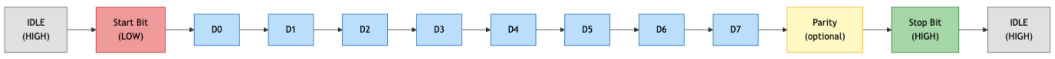


Figure 1.1 *– UART serial transmission frame structure*

On the Arduino Mega 2560, UART0 is connected to the USB-to-Serial converter chip, allowing seamless communication with the host PC over a USB cable.

C Standard I/O Library (STDIO)

The C Standard I/O library (<`stdio.h`>) provides portable, high-level functions for formatted text input and output. In desktop environments, these functions interact with console I/O; in embedded systems, they can be redirected to any I/O peripheral.

The key STDIO functions used in this lab are `printf()`, which provides formatted text output to `stdout` for displaying prompts, confirmation messages, and error messages, and `fgets()`, which reads a line of text from `stdin` to capture user commands from the serial terminal.

On AVR microcontrollers with the Arduino framework, STDIO streams (`stdout`, `stdin`) are not connected to any device by default. The AVR libc library provides the `fdev_setup_stream()` function to create custom FILE streams backed by user-defined read/write functions. By assigning a custom stream to `stdout` and `stdin`, all subsequent calls to `printf()` and `fgets()` are transparently redirected to the serial port.

PlatformIO Build System

PlatformIO is a professional open-source ecosystem for embedded development. It provides project management, library dependency resolution, multi-board support, and integrated build/upload/monitor tools. In this lab, PlatformIO is used within VS Code to compile, upload, and monitor the application on the Arduino Mega 2560.

Wokwi Simulator

Wokwi is an online and VS Code-integrated simulator for embedded systems. It supports Arduino, ESP32, and other popular platforms. Wokwi allows testing of the application in a virtual environment before deploying to physical hardware, providing a simulated serial terminal and virtual LEDs.

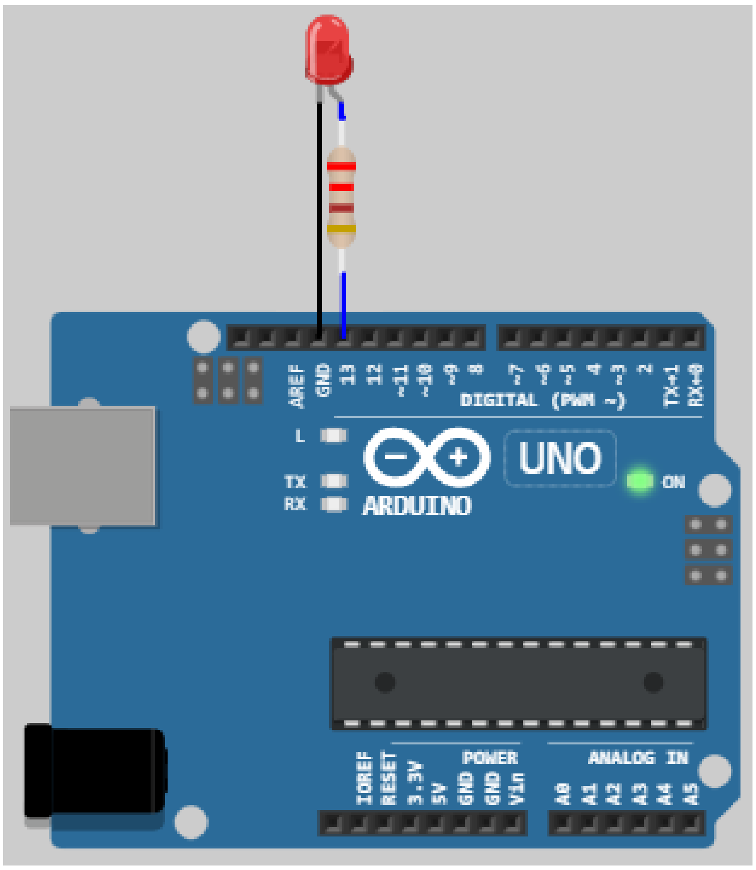


Figure 1.2 *– Example of circuit simulation in Wokwi with Arduino and LED*

In this laboratory work, Wokwi is integrated with PlatformIO through a configuration file (`wokwi.toml`). This integration allows the compiled firmware to be executed in the simulator for testing and verification without physical hardware, as shown in *Figure 1.2*.

1.4. Hardware Components

Arduino Mega 2560

The Arduino Mega 2560 is a microcontroller development board based on the ATmega2560 AVR chip. It features 54 digital I/O pins (15 of which support PWM output), 16 analog inputs, 4 hardware UART ports, a 16 MHz crystal oscillator, a USB connection, a power jack, and an ICSP header.

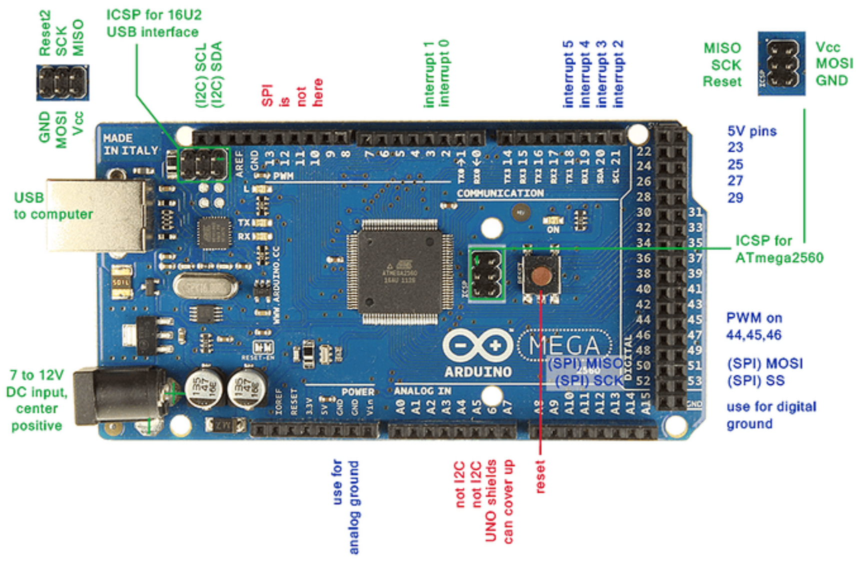


Figure 1.3 – *Arduino Mega 2560 development board with AVR architecture*

The Arduino Mega 2560 board, shown in *Figure 1.3*, uses an ATmega2560 microcontroller based on the AVR architecture. This 8-bit RISC processor operates at 16 MHz. Key specifications relevant to this lab:

- **Microcontroller:** ATmega2560 (8-bit AVR, 16 MHz)
- **Flash memory:** 256 KB (8 KB used by bootloader)
- **SRAM:** 8 KB
- **Digital I/O pins:** 54
- **UART ports:** 4 (UART0 connected to USB)
- **Operating voltage:** 5V
- **I/O pin output current:** 20 mA (max 40 mA)

UART0 (pins 0/TX and 1/RX) is connected through a USB-to-Serial converter chip, enabling direct communication with the host PC for programming and serial monitoring.

LED (Light-Emitting Diode)

A standard 5mm red LED is used as the output actuator in this lab. The LED is connected to digital pin 7 of the Arduino Mega through a current-limiting resistor. When pin 7 is set HIGH (5V), current flows through the resistor and LED, causing it to emit light. When set LOW (0V), the LED turns off.

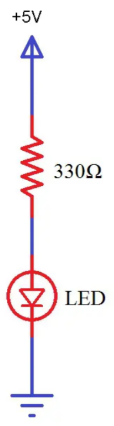


Figure 1.4 *– Electrical connection circuit of the LED to the microcontroller*

LED specifications:

- **Forward voltage (Vf):** approximately 1.8–2.2V (red LED)
- **Forward current (If):** 20 mA (typical operating current)
- **Maximum current:** 30 mA

220 Ω Resistor

A 220 Ω resistor is placed in series with the LED to limit the current flowing through it. The resistor value is calculated based on Ohm's law:

$$I = \frac{V_{pin} - V_{LED}}{R} = \frac{5V - 2V}{220\Omega} \approx 13.6mA$$

This current is within the safe operating range of both the LED (< 20 mA) and the Arduino GPIO pin (< 40 mA absolute maximum).

Breadboard and Jumper Wires

A solderless breadboard provides a convenient prototyping platform for connecting the LED, resistor, and Arduino without soldering. Jumper wires are used to make electrical connections between the Arduino pins, the resistor, the LED, and the ground rail.

1.5. Software Components

Visual Studio Code with PlatformIO

Visual Studio Code (VS Code) is a lightweight but powerful source code editor. The PlatformIO IDE extension transforms it into a full-featured embedded development environment by providing project initialization and configuration via `platformio.ini`, automatic toolchain and library management, code compilation, firmware upload, and serial monitor capabilities, as well as IntelliSense code completion for Arduino/AVR libraries.

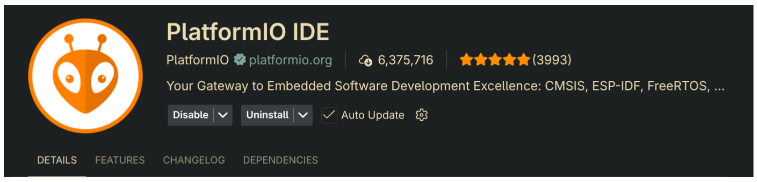

PlatformIO IDE
PlatformIO platformio.org
6,375,716
(3993)
Your Gateway to Embedded Software Development Excellence: CMSIS, ESP-IDF, FreeRTOS, ...
Disable
Uninstall
Auto Update
DETAILS
FEATURES
CHANGELOG
DEPENDENCIES

Figure 1.5 *– PlatformIO IDE extension in Visual Studio Code*

For this project, PlatformIO IDE is used to configure the Arduino Mega 2560 development board, manage the project structure (including `src/`, `lib/`, and `include/` directories), and compile the source code targeting the ATmega2560 microcontroller. The `platformio.ini` configuration file specifies the board type, framework, and build settings, as shown in *Figure 1.5*. PlatformIO handles the compilation, linking, and firmware upload process, while its library manager facilitates the integration of custom libraries such as the LED driver and STDIO serial service used in this application.

Wokwi VS Code Extension

The Wokwi VS Code extension enables hardware simulation directly within the IDE. It reads a `diagram.json` file defining the virtual circuit and a `wokwi.toml` file pointing to the compiled firmware. This allows interactive testing of the serial command interface without physical hardware.

Serial Terminal (PlatformIO Monitor)

PlatformIO's built-in serial monitor provides a terminal interface for sending and receiving text over the UART connection. It operates at a configurable baud rate (9600 in this lab) and displays characters received from the microcontroller while transmitting typed characters back to it.

1.6. System Architecture and Justification

The system follows a layered architecture that organizes functionality into distinct levels, each with specific responsibilities. This structure separates concerns and enables modular development. The APP (Application Layer) implements command processing logic and high-level functionalities. The SRV (Service Layer) provides common services such as command parsing and data formatting. The ECAL (ECU Abstraction Layer) abstracts platform-specific hardware details, including STDIO-to-UART redirection. The MCAL (Microcontroller Abstraction Layer) interfaces with microcontroller peripherals such as LED GPIO control. Finally, the HW (Hardware) layer represents the physical ATmega2560 microcontroller and its hardware peripherals.

This layered organization enables independent testing of components and allows modifications or extensions without affecting other system parts. The hardware-software interface operates across multiple abstraction layers, facilitating clear separation between application logic and hardware control.

1.7. Case Study: Serial Command Interfaces in Embedded Systems

Serial command-line interfaces (CLIs) are widely used in embedded systems for diagnostics, configuration, and debugging. Network equipment such as routers and switches (e.g., Cisco IOS, OpenWrt) use serial consoles for initial configuration and troubleshooting. Industrial PLCs (Programmable Logic Controllers) often accept serial commands for parameter tuning and status queries. IoT gateways typically provide serial debug interfaces for firmware updates and log inspection. Consumer devices like 3D printers use G-code commands over serial (typically UART) to control motion, temperature, and extrusion.

The STDIO-based approach demonstrated in this lab mirrors these real-world implementations, where a standardized text protocol enables human-machine interaction through a simple terminal connection. The modular architecture (separating I/O abstraction from application logic) is a fundamental pattern in professional embedded firmware development.

2. Design

2.1. System Architecture Diagrams

Component-Level Architecture

The system consists of three main components: a host PC running a serial terminal, the Arduino Mega 2560 microcontroller, and the LED output circuit. The PC communicates with the MCU over a USB-to-UART bridge, sending text commands and receiving confirmation messages. The MCU processes these commands and drives the LED accordingly.

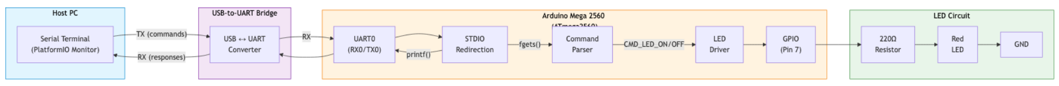


Figure 2.1 *– System structural diagram showing the PC, MCU, and LED circuit*

The system structural diagram illustrates the data flow:

1. The user types a command (e.g., `led on`) in the serial terminal on the PC.
2. The command is transmitted over USB, converted to UART by the USB-to-Serial bridge on the Arduino board, and received by the ATmega2560's UART0 peripheral.
3. The STDIO library redirects the incoming bytes to `stdin`, allowing `fgets()` to read the complete command line.
4. The command parser module interprets the text and determines the action.

5. The LED driver module sets the GPIO pin HIGH or LOW to control the LED.
6. A confirmation message is sent back via `printf()` through `stdout`, which is redirected to UART TX, travels back over USB, and appears in the serial terminal.

Layered System Architecture

The software follows a layered architecture that separates concerns:

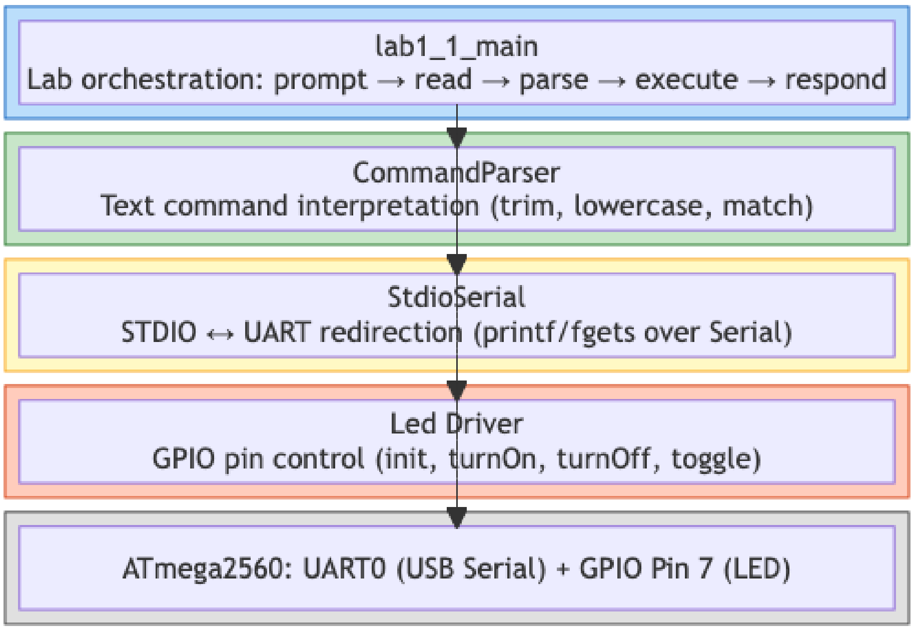

lab1_1_main
Lab orchestration: prompt → read → parse → execute → respond
CommandParser
Text command interpretation (trim, lowercase, match)
StdioSerial
STDIO ↔ UART redirection (printf/fgets over Serial)
Led Driver
GPIO pin control (init, turnOn, turnOff, toggle)
ATmega2560: UART0 (USB Serial) + GPIO Pin 7 (LED)

Figure 2.2 *– Layered software architecture of the system*

The software follows a layered architecture that separates concerns as illustrated in *Figure 2.2*. The Application Layer (APP) contains `lab1_1_main`, which orchestrates the main control loop by prompting for input, reading commands, invoking the parser, executing LED actions, and displaying feedback. The Service Layer (SRV) contains `CommandParser`, which provides command interpretation as a reusable service that trims whitespace, normalizes case, and matches input strings against known commands. The ECU Abstraction Layer (ECAL) contains `StdioSerial`, which abstracts the UART hardware into C standard I/O streams by creating a bridge between the portable STDIO API and the Arduino Serial hardware. The Microcontroller Abstraction Layer (MCAL) contains the `Led` driver, which provides a clean object-oriented API for GPIO-based LED control, hiding direct `pinMode()`/`digitalWrite()` calls from upper layers. Finally, the Hardware (HW) layer consists of the physical ATmega2560 MCU with its UART0 peripheral (connected to USB) and digital GPIO pin 7 (connected to the LED circuit).

2.2. Block Diagrams

The application algorithm is divided into two phases: initialization and the main command loop.

Initialization Phase

During startup, the system configures the STDIO serial redirection (UART at 9600 baud, `stdout/stdin` redirected), initializes the LED driver (pin 7 as OUTPUT, LED OFF), and prints a welcome banner listing the available commands.

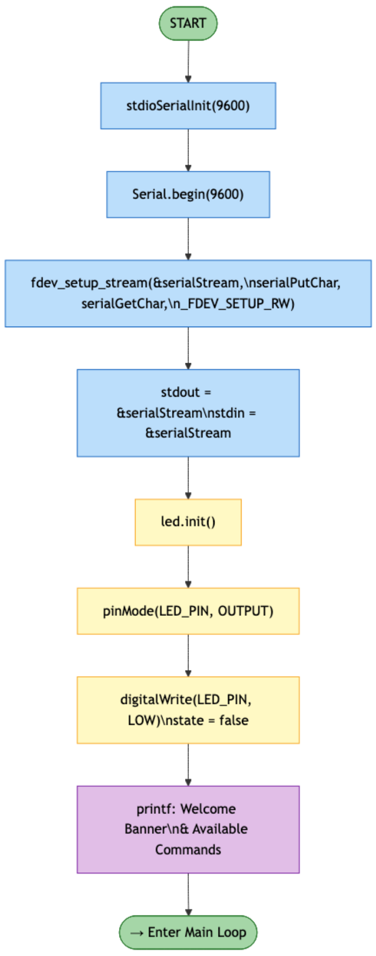

START
stdioSerialInit(9600)
Serial.begin(9600)
fdev_setup_stream(&serialStream,\nserialPutChar, serialGetChar,\n_FDEV_SETUP_RW)
stdout = &serialStream\nstdin = &serialStream
led.init()
pinMode(LED_PIN, OUTPUT)
digitalWrite(LED_PIN, LOW)\nstate = false
printf: Welcome Banner\n& Available Commands
→ Enter Main Loop

Figure 2.3 *– Flowchart of the initialization phase*

Main Command Loop

After initialization, the system enters an infinite loop: it displays a prompt, waits for a line from `stdin` (blocking `fgets()`), parses the command, executes the corresponding LED action, and prints a confirmation or error message.

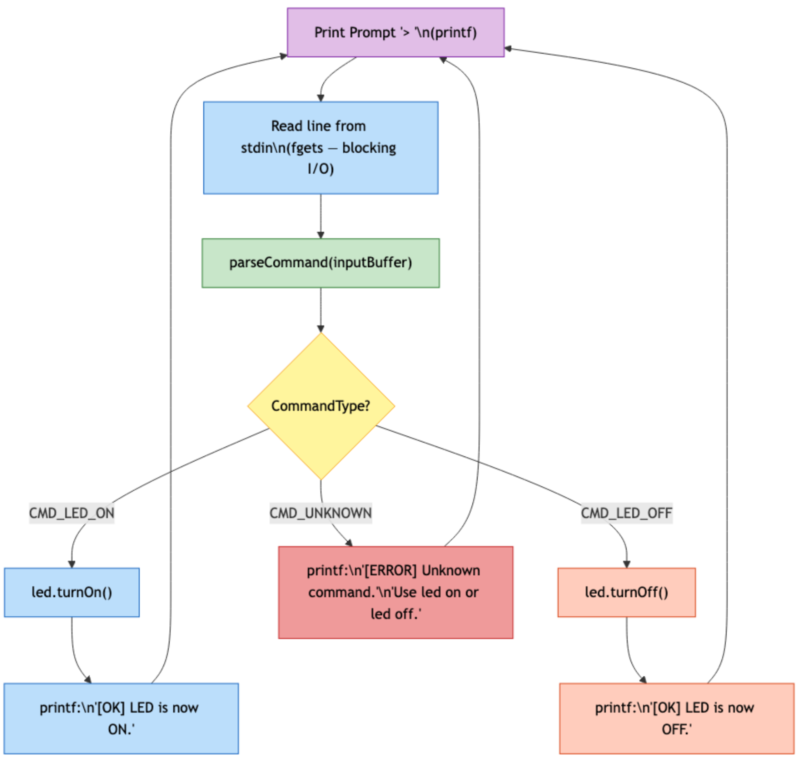

Print Prompt '> '\n(printf)
Read line from stdin\n(fgets – blocking I/O)
parseCommand(inputBuffer)
CommandType?
CMD_LED_ON
CMD_UNKNOWN
CMD_LED_OFF
led.turnOn()
printf:\n'[ERROR] Unknown command.'\n'Use led on or led off.'
led.turnOff()
printf:\n'[OK] LED is now ON.'
printf:\n'[OK] LED is now OFF.'

Figure 2.4 *– Flowchart of the main command processing loop*

2.3. Electrical Schematics

The circuit is straightforward: a single red LED is connected in series with a 220 Ω current-limiting resistor between digital pin 7 of the Arduino Mega and ground.

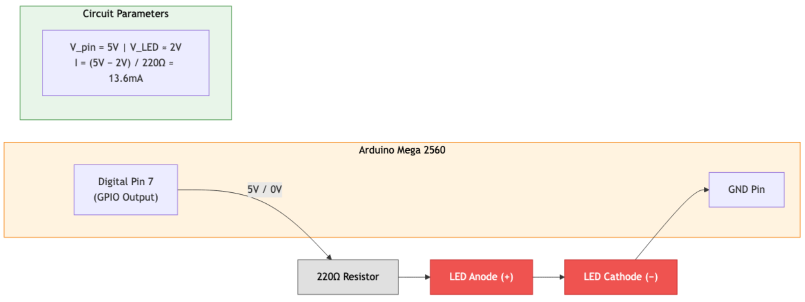


Figure 2.5 *– Electrical schematic — LED control circuit*

Component Specification

The circuit consists of three main components. The Arduino Mega 2560 is a microcontroller board based on the ATmega2560 with 5V logic and a 16 MHz clock. It provides the digital GPIO output (pin 7) used to control the LED and the UART0 serial interface (connected via USB). A red LED with a 5mm diameter serves as the output indicator, with a forward voltage of approximately 2.0 V and a maximum forward current rating of 20 mA. A 220 Ω resistor acts as the current-limiting component, reducing the LED current to approximately 13.6 mA at the 5V supply voltage.

Circuit Connections

The circuit forms a simple series path for current flow. Arduino Pin 7 connects to the first terminal of the 220 Ω resistor, with the GPIO output driving current through the resistor. The second resistor terminal connects to the LED anode (positive terminal), allowing current to flow from the resistor into the LED. The LED cathode (negative terminal) connects back to the Arduino ground, completing the circuit. When pin 7 outputs HIGH (5V), current $I = (5V - 2V)/220\Omega \approx 13.6mA$ flows through the LED, illuminating it. When pin 7 outputs LOW (0V), no current flows and the LED is off.

Hardware Configuration

The Wokwi simulation implements the above circuit virtually. *Figure 2.6* shows the assembled circuit in the Wokwi simulator.

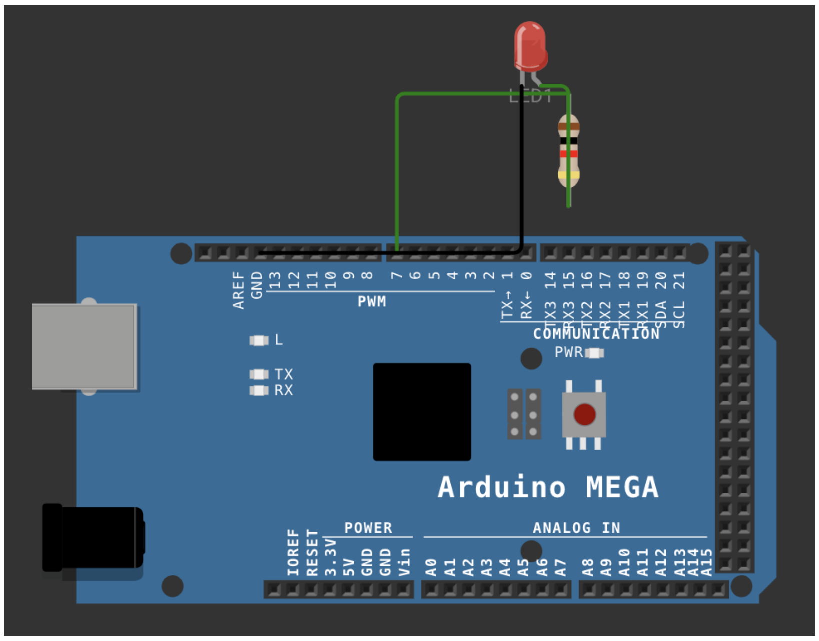


Figure 2.6 *– Wokwi simulation circuit — Arduino Mega 2560 with LED and 220 Ω resistor*

The Wokwi simulation configuration file defines the virtual circuit:

Listing 2.1 *– Wokwi diagram.json — Virtual circuit definition*

```
{
  "version": 1,
  "author": "Vremere Adrian",
  "editor": "wokwi",
  "parts": [
    {
      "type": "wokwi-arduino-mega",
      "id": "mega",
      "top": 0, "left": 0, "attrs": {}
    },
    {
      "type": "wokwi-resistor",
      "id": "r1",
      "top": -50, "left": 250, "rotate": 90,
      "attrs": { "resistance": "220" }
    },
    {
      "type": "wokwi-led",
      "id": "led1",
      "top": -120, "left": 240,
      "attrs": { "color": "red", "label": "LED1" }
    }
  ],
  "connections": [
    ["mega:7", "r1:1", "green", ["v0"]],
    ["r1:2", "led1:A", "green", ["v0"]],
    ["led1:C", "mega:GND.1", "black", ["v0"]]
  ]
}
```

2.4. Project Structure

The project follows a modular directory layout as required by the course conventions:

***Listing 2.2** – Project directory structure for Lab 1.1*

```
labs/
|-- platformio.ini               # Build config (env:lab1_1)
|-- src/
|   |-- main.cpp                 # Entry point (lab selector)
|-- lab/
|   |-- lab1_1/
|       |-- lab1_1_main.h        # Lab 1.1 interface
|       |-- lab1_1_main.cpp      # Lab 1.1 implementation
|-- lib/
|   |-- Led/
|   |   |-- Led.h                # LED driver interface
|   |   |-- Led.cpp              # LED driver implementation
|   |-- StdioSerial/
|   |   |-- StdioSerial.h        # STDIO redirection interface
|   |   |-- StdioSerial.cpp      # STDIO redirection implementation
|   |-- CommandParser/
|       |-- CommandParser.h      # Command parser interface
|       |-- CommandParser.cpp    # Command parser implementation
|-- wokwi/
    |-- lab1.1/
        |-- diagram.json         # Wokwi circuit definition
        |-- wokwi.toml           # Wokwi firmware config
```

Each library under `lib/` is a self-contained, reusable module with a clear interface (`.h`) and implementation (`.cpp`). The lab-specific code resides in `lab/lab1_1/`, while `src/main.cpp` simply delegates to the active lab's setup and loop functions.

2.5. Modular Implementation

MCAL Layer: Led Driver

The Led driver provides a hardware abstraction for controlling an LED connected to a GPIO pin. It encapsulates `pinMode()` and `digitalWrite()` behind a clean object-oriented interface.

Listing 2.3 *– Led.h — LED driver interface*

```
#ifndef LED_H
#define LED_H

#include <Arduino.h>

class Led {
public:
    Led(uint8_t pin);
    void init();
    void turnOn();
    void turnOff();
    void toggle();
    bool isOn() const;
private:
    uint8_t ledPin;
    bool state;
};

#endif
```

Listing 2.4 *– Led.cpp — LED driver implementation*

```
#include "Led.h"

Led::Led(uint8_t pin) : ledPin(pin), state(false) {}

void Led::init() {
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
    state = false;
}

void Led::turnOn() {
    digitalWrite(ledPin, HIGH);
    state = true;
}
```

```
void Led::turnOff() {
    digitalWrite(ledPin, LOW);
    state = false;
}

void Led::toggle() {
    if (state) { turnOff(); }
    else { turnOn(); }
}

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

The `Led` class accepts a pin number at construction time, making it reusable for any GPIO-connected LED. The `init()` method must be called once during setup to configure the pin direction.

The `init()` function initializes the LED pin by configuring it as an output and setting the initial state to OFF, as illustrated in *Figure 2.7*. The function ensures each LED is properly configured before application execution begins.

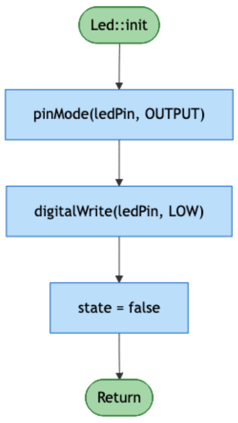


Figure 2.7 *– Flowchart: LED driver initialization*

The `turnOn()` function's operational flow is shown in *Figure 2.8*, demonstrating the GPIO state modification to illuminate the LED.

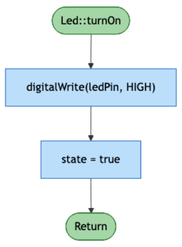


Figure 2.8 *– Flowchart: LED activation function*

The `turnOff()` function operates symmetrically, setting the pin LOW and updating the state, as shown in *Figure 2.9*.

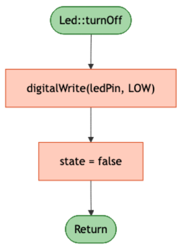


***Figure 2.9** – Flowchart: LED deactivation function*

This driver demonstrates proper abstraction layer design: application code interacts with logical LED methods rather than physical pin numbers, hardware dependencies are isolated in the constructor, and the implementation remains independent of the specific pin assignments.

ECAL Layer: StdioSerial

The StdioSerial module redirects the C standard I/O streams (`stdout`, `stdin`) to the hardware UART, enabling the use of `printf()` and `fgets()` for serial communication.

***Listing 2.5** – StdioSerial.h — STDIO redirection interface*

```
#ifndef STDIO_SERIAL_H
#define STDIO_SERIAL_H

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

void stdioSerialInit(unsigned long baudRate);

#endif
```

***Listing 2.6** – StdioSerial.cpp — STDIO redirection implementation (key sections)*

```
#include "StdioSerial.h"

static int serialPutChar(char c, FILE *stream) {
    Serial.write(c);
    return 0;
}

static int serialGetChar(FILE *stream) {
    while (!Serial.available()) { }
    char c = Serial.read();
    if (c == '\r') {
        Serial.write('\r');
        Serial.write('\n');
        return '\n';
    }
    Serial.write(c); // Echo
    return c;
}

static FILE serialStream;

void stdioSerialInit(unsigned long baudRate) {
    Serial.begin(baudRate);
    while (!Serial) { ; }
    fdev_setup_stream(&serialStream,
        serialPutChar, serialGetChar, _FDEV_SETUP_RW);
    stdout = &serialStream;
    stdin  = &serialStream;
}
```

The implementation uses AVR libc's `fdev_setup_stream()` to create a custom `FILE` stream. The `serialPutChar()` function writes characters to the UART (used by `printf`), while `serialGetChar()` reads characters with local echo and carriage-return-to-newline conversion (used by `fgets`).

The `stdioSerialInit()` function performs three essential operations: initializing the UART hardware at the specified baud rate, associating custom I/O functions with a file stream using `fdev_setup_stream()`, and redirecting standard I/O descriptors to this stream. The operational flow is illustrated in *Figure 2.10*.

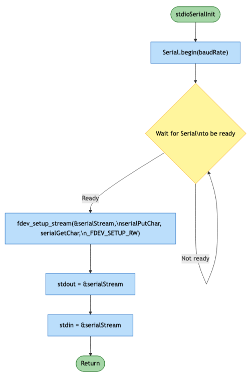

stdioSerialInit
Serial.begin(baudRate)
Wait for Serial\nto be ready
Ready
Not ready
fdev_setup_stream(&serialStream,\nserialPutChar, serialGetChar,\n_FDEV_SETUP_RW)
stdout = &serialStream
stdin = &serialStream
Return

Figure 2.10 *– Flowchart: Serial STDIO initialization process*

Character transmission (`serialPutChar()`) places data in the UART transmit buffer, as shown in *Figure 2.11*. The function writes a single character to the serial port and returns immediately after queuing the data.

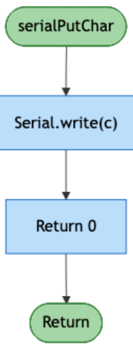


Figure 2.11 *– Flowchart: Character transmission to serial port*

Character reception (`serialGetChar()`) implements blocking read with character echo for user feedback, as depicted in *Figure 2.12*. The function waits until data becomes available in the receive buffer, reads the character, echoes it back to the terminal for visual confirmation, and returns the received value.

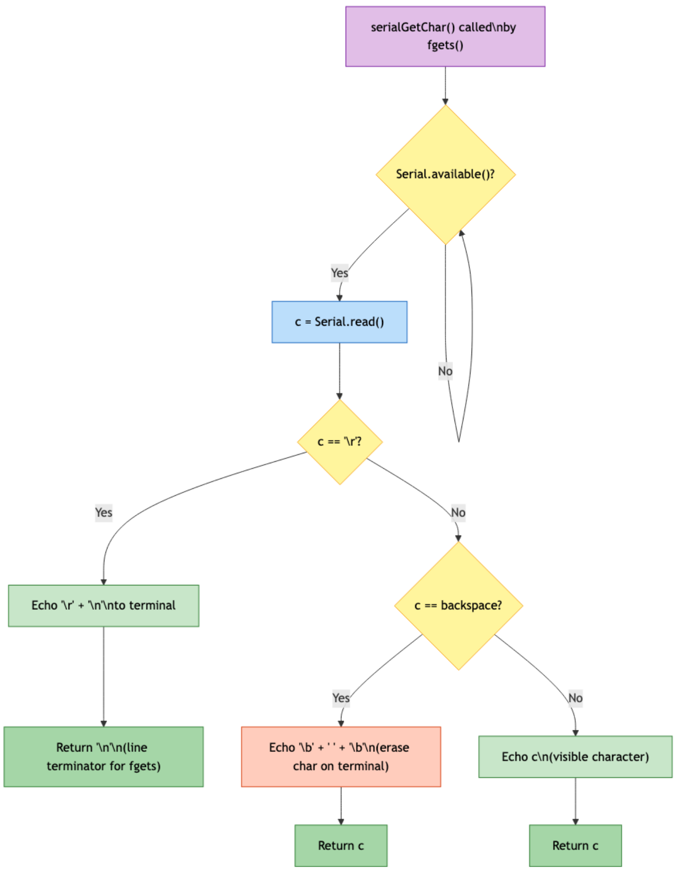

serialGetChar() called\nby fgets()
Serial.available()?
Yes
No
c = Serial.read()
c == '\r'?
Yes
No
Echo '\r' + '\n'\nto terminal
c == backspace?
Yes
No
Return '\n'\n(line terminator for fgets)
Echo '\b' + ' ' + '\b'\n(erase char on terminal)
Echo c\n(visible character)
Return c
Return c

Figure 2.12 *– Flowchart: Character reception from serial port*

This service layer eliminates direct hardware manipulation from application code, enabling portable I/O operations through standard library functions.

SRV Layer: CommandParser

The CommandParser module interprets text strings and maps them to command types.

Listing 2.7 – CommandParser.h — Command parser interface

```cpp
#ifndef COMMAND_PARSER_H
#define COMMAND_PARSER_H

enum CommandType {
    CMD_UNKNOWN,
    CMD_LED_ON,
    CMD_LED_OFF
};

CommandType parseCommand(const char *input);

#endif
```

Listing 2.8 – CommandParser.cpp — Command parser implementation (key sections)

```cpp
#include "CommandParser.h"
#include <string.h>
#include <ctype.h>

static void toLowerStr(char *dest,
    const char *src, size_t maxLen) { /* ... */ }

static void trimStr(char *dest,
    const char *src, size_t maxLen) { /* ... */ }

CommandType parseCommand(const char *input) {
    char trimmed[64], lower[64];
    trimStr(trimmed, input, sizeof(trimmed));
    toLowerStr(lower, trimmed, sizeof(lower));

    if (strcmp(lower, "led on") == 0)
        return CMD_LED_ON;
    if (strcmp(lower, "led off") == 0)
        return CMD_LED_OFF;

    return CMD_UNKNOWN;
}
```

The parser performs two preprocessing steps (trim whitespace, convert to lowercase) before matching against known commands. This ensures robust handling of user input variations (leading/trailing spaces, mixed case).

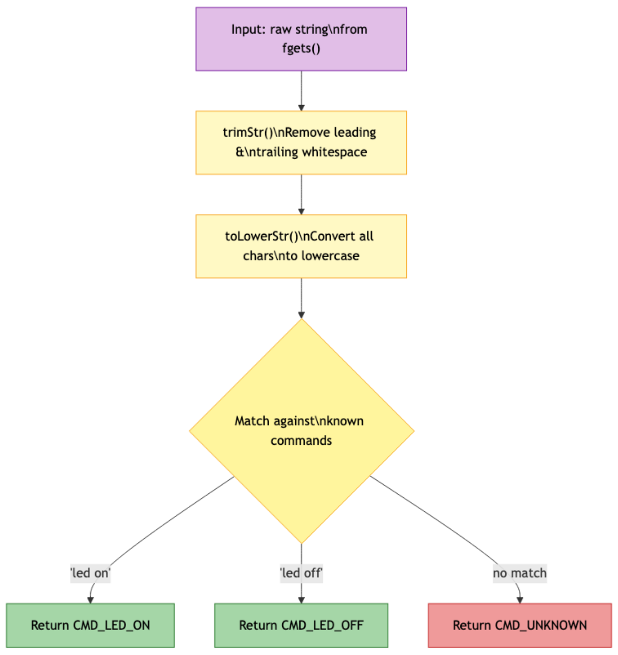

Input: raw string\nfrom fgets()
trimStr()\nRemove leading &\ntrailing whitespace
toLowerStr()\nConvert all chars\nto lowercase
Match against\nknown commands
'led on'
'led off'
no match
Return CMD_LED_ON
Return CMD_LED_OFF
Return CMD_UNKNOWN

Figure 2.13 *– Functional flowchart of the CommandParser module*

APP Layer: Lab 1.1 Main

The application layer ties all modules together:

Listing 2.9 *– lab1_1_main.cpp — Application entry point (key sections)*

```
#include "lab1_1_main.h"
#include <stdio.h>
#include "Led.h"
#include "StdioSerial.h"
#include "CommandParser.h"

static const uint8_t LED_PIN = 7;
static const unsigned long BAUD_RATE = 9600;
static Led led(LED_PIN);
static char inputBuffer[64];

void lab1_1Setup() {
    stdioSerialInit(BAUD_RATE);
    led.init();
    printf("\r\n=== Lab 1.1: Serial LED Control ===\r\n");
    printf("Commands: led on, led off\r\n");
}

void lab1_1Loop() {
    printf("> ");
    if (fgets(inputBuffer, sizeof(inputBuffer), stdin)) {
        CommandType cmd = parseCommand(inputBuffer);
        switch (cmd) {
            case CMD_LED_ON:
                led.turnOn();
                printf("[OK] LED is now ON.\r\n");
                break;
            case CMD_LED_OFF:
                led.turnOff();
                printf("[OK] LED is now OFF.\r\n");
                break;
            case CMD_UNKNOWN:
                printf("[ERROR] Unknown command.\r\n");
                break;
        }
    }
}
```

The application logic is clean and readable: initialize peripherals, then loop over prompt-read-parse-execute-respond. All I/O uses standard C functions (`printf`, `fgets`), fulfilling the laboratory requirement.

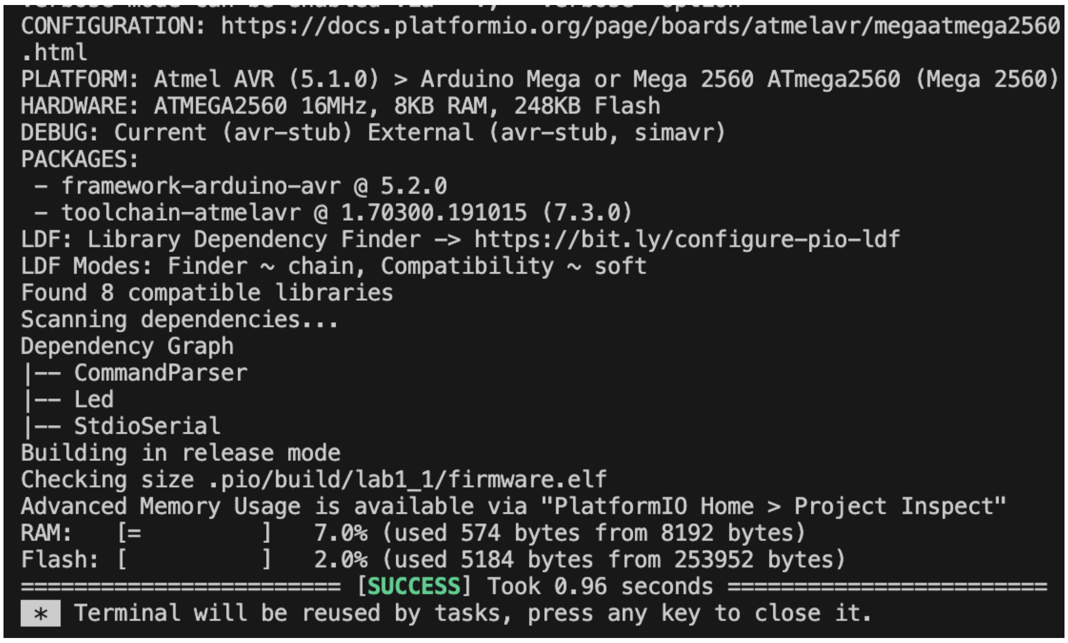
The module's setup function establishes the execution environment by initializing dependencies in proper order: serial communication first (enabling `printf`/`fgets`), then LED hardware. The main loop implements the command processing algorithm whose operational flow and detailed step-by-step execution were thoroughly described in *Figure 2.4*. The implementation uses standard library functions for I/O operations and the Led class interface for hardware control, maintaining complete independence from hardware-specific details.

This architectural approach ensures that hardware modifications propagate only to the relevant abstraction layer—changing LED pin assignments requires only modifying the constructor argument, while replacing the serial interface would affect only `StdioSerial`. The application layer remains entirely unaffected by such changes, demonstrating the power of proper abstraction boundaries.

3. Obtained Results

3.1. Build Process

The project was compiled using PlatformIO with the `lab1_1` environment targeting the Arduino Mega 2560 (ATmega2560). The build completed successfully with no errors or warnings.



```
CONFIGURATION: https://docs.platformio.org/page/boards/atmelavr/megaatmega2560
.html
PLATFORM: Atmel AVR (5.1.0) > Arduino Mega or Mega 2560 ATmega2560 (Mega 2560)
HARDWARE: ATMEGA2560 16MHz, 8KB RAM, 248KB Flash
DEBUG: Current (avr-stub) External (avr-stub, simavr)
PACKAGES:
 - framework-arduino-avr @ 5.2.0
 - toolchain-atmelavr @ 1.70300.191015 (7.3.0)
LDF: Library Dependency Finder -> https://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 8 compatible libraries
Scanning dependencies...
Dependency Graph
|-- CommandParser
|-- Led
|-- StdioSerial
Building in release mode
Checking size .pio/build/lab1_1/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [=         ]   7.0% (used 574 bytes from 8192 bytes)
Flash: [          ]   2.0% (used 5184 bytes from 253952 bytes)
======================== [SUCCESS] Took 0.96 seconds ========================
 *  Terminal will be reused by tasks, press any key to close it.
```

Figure 3.1 – *Successful PlatformIO build output for Lab 1.1*

3.2. Simulation Initialization

Upon starting the Wokwi simulation, the virtual Arduino Mega 2560 boots and the application initializes the STDIO serial redirection and LED driver. The serial terminal displays the welcome banner and command prompt:

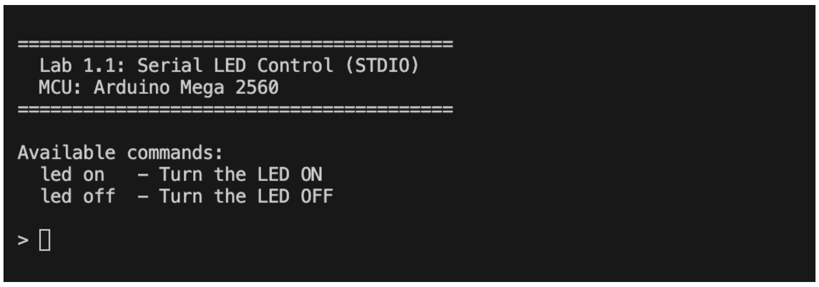
Listing 3.1 – *Expected serial output on startup*

```
========================================
  Lab 1.1: Serial LED Control (STDIO)
  MCU: Arduino Mega 2560
========================================

Available commands:
  led on   - Turn the LED ON
  led off  - Turn the LED OFF

>
```

The LED starts in the OFF state (no current flowing through the circuit).



```
========================================
  Lab 1.1: Serial LED Control (STDIO)
  MCU: Arduino Mega 2560
========================================

Available commands:
  led on   - Turn the LED ON
  led off  - Turn the LED OFF

> 
```

Figure 3.2 – *Initial simulation state — LED off, welcome banner displayed*

3.3. Command Interface Validation

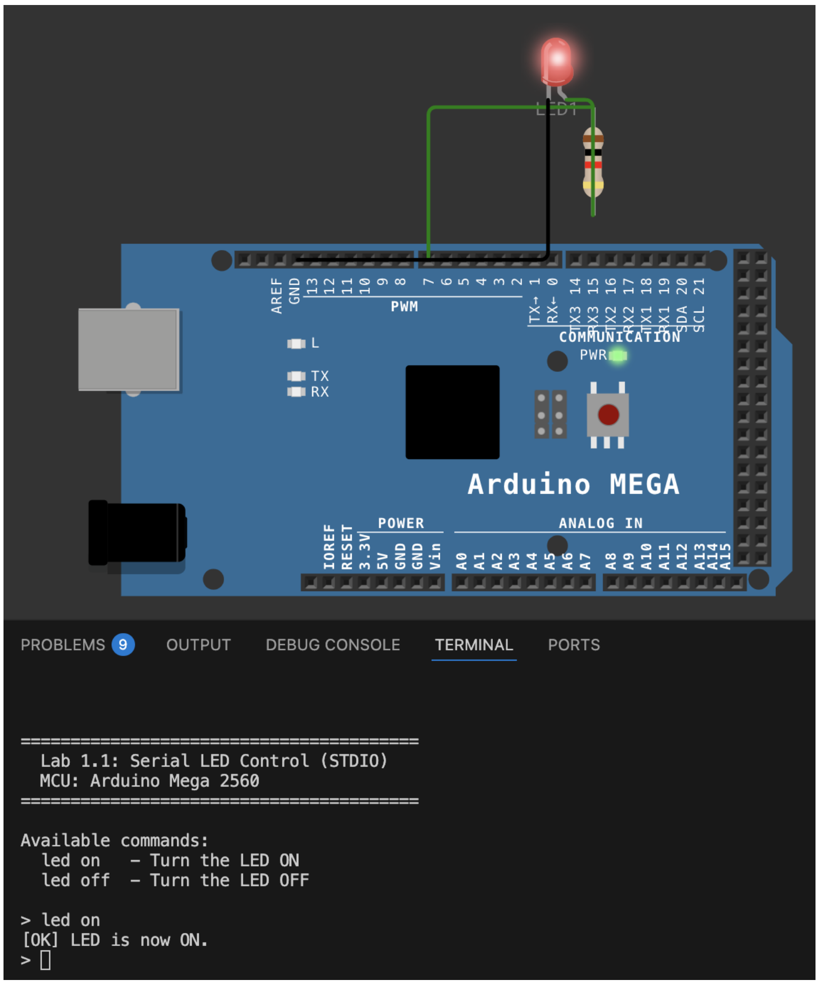
Test: LED ON Command

Typing `led on` and pressing Enter turns the LED on. The system responds with a confirmation message.

Listing 3.2 – Serial output — LED ON command

```
> led on
[OK] LED is now ON.
>
```

The LED in the Wokwi simulation lights up (red), confirming that pin 7 is driven HIGH.



```
========================================
  Lab 1.1: Serial LED Control (STDIO)
  MCU: Arduino Mega 2560
========================================

Available commands:
  led on   - Turn the LED ON
  led off  - Turn the LED OFF

> led on
[OK] LED is now ON.
> 
```

Figure 3.3 – LED ON — simulation result with serial confirmation

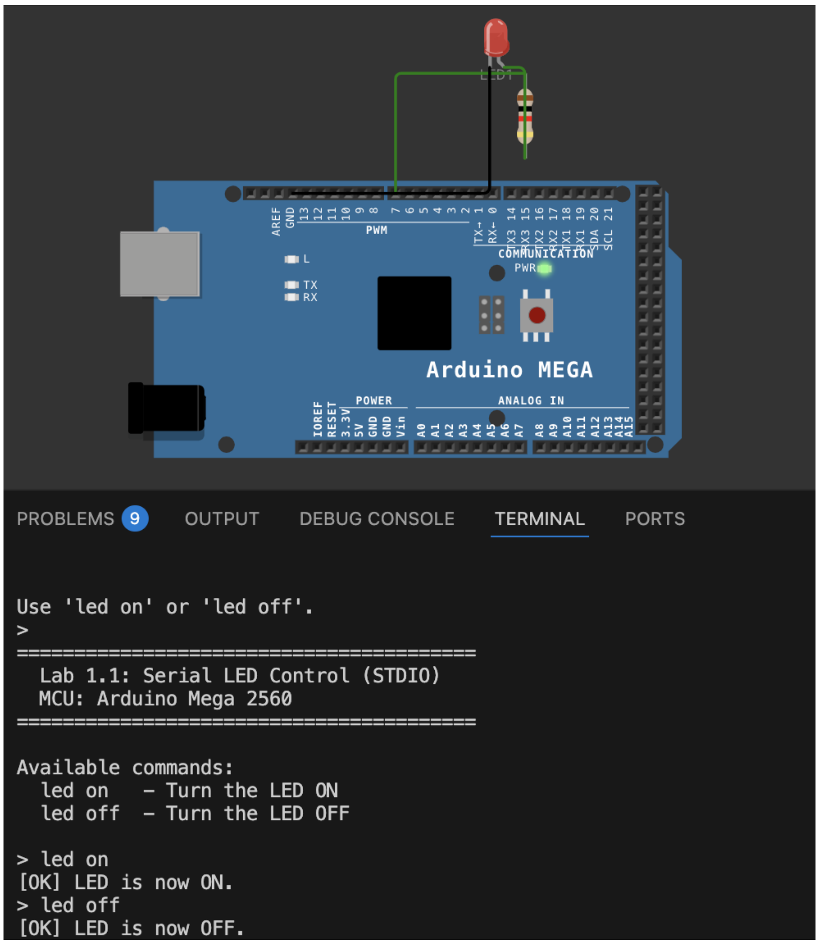
Test: LED OFF Command

Typing `led off` turns the LED off, and the system confirms:

Listing 3.3 – Serial output — LED OFF command

```
> led off
[OK] LED is now OFF.
```

```
>
```



```
Use 'led on' or 'led off'.
>
========================================
  Lab 1.1: Serial LED Control (STDIO)
  MCU: Arduino Mega 2560
========================================

Available commands:
  led on   - Turn the LED ON
  led off  - Turn the LED OFF

> led on
[OK] LED is now ON.
> led off
[OK] LED is now OFF.
```

Figure 3.4 – LED OFF — simulation result with serial confirmation

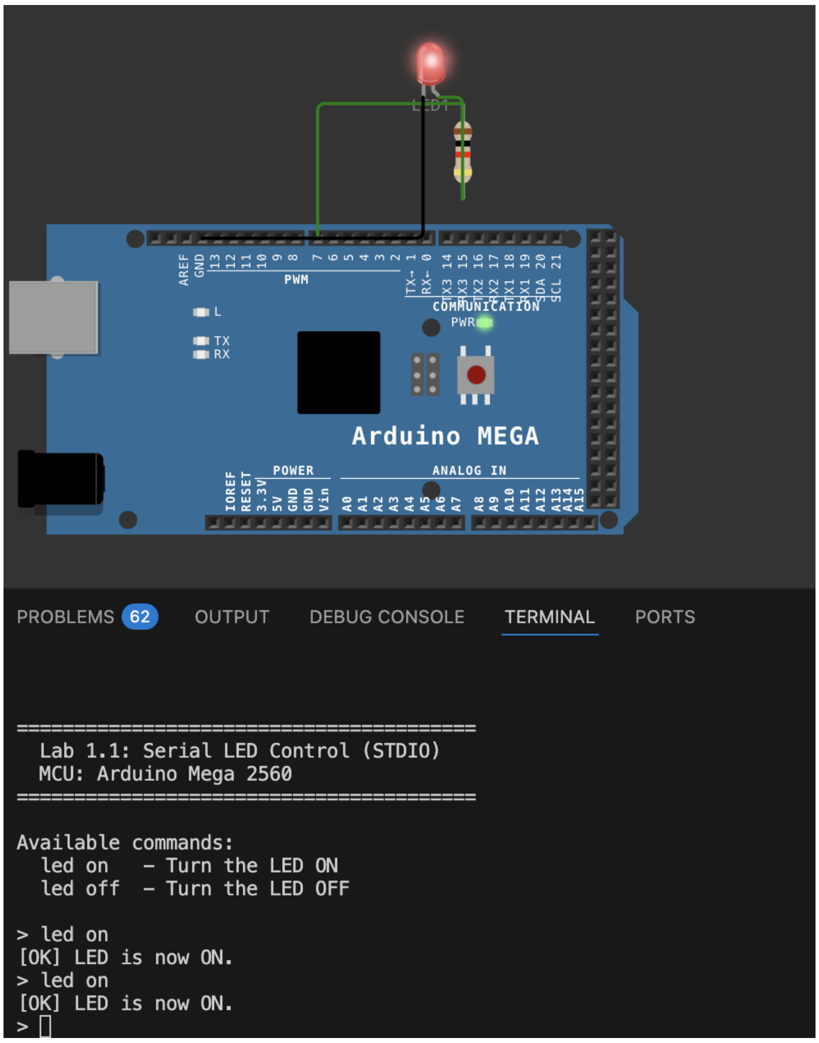
Test: Command Persistence

An important observation emerges when repeatedly issuing the same command. Sending `led on` while the LED is already illuminated maintains its on state without any visible change:

Listing 3.4 – Serial output — repeated LED ON command

```
> led on
[OK] LED is now ON.
> led on
[OK] LED is now ON.
>
```

Similarly, sending `led off` when the LED is already off still maintains the off state. This behavior is expected and demonstrates the unidirectional nature of the LED control interface. The microcontroller sends control signals to the LED but does not receive feedback about its current state. The system operates in open-loop control mode—commands are executed unconditionally without verifying the LED's previous state.



```
========================================
  Lab 1.1: Serial LED Control (STDIO)
  MCU: Arduino Mega 2560
========================================

Available commands:
  led on   - Turn the LED ON
  led off  - Turn the LED OFF

> led on
[OK] LED is now ON.
> led on
[OK] LED is now ON.
> 
```

***Figure 3.5** – Repeated `led on` command maintains LED state*

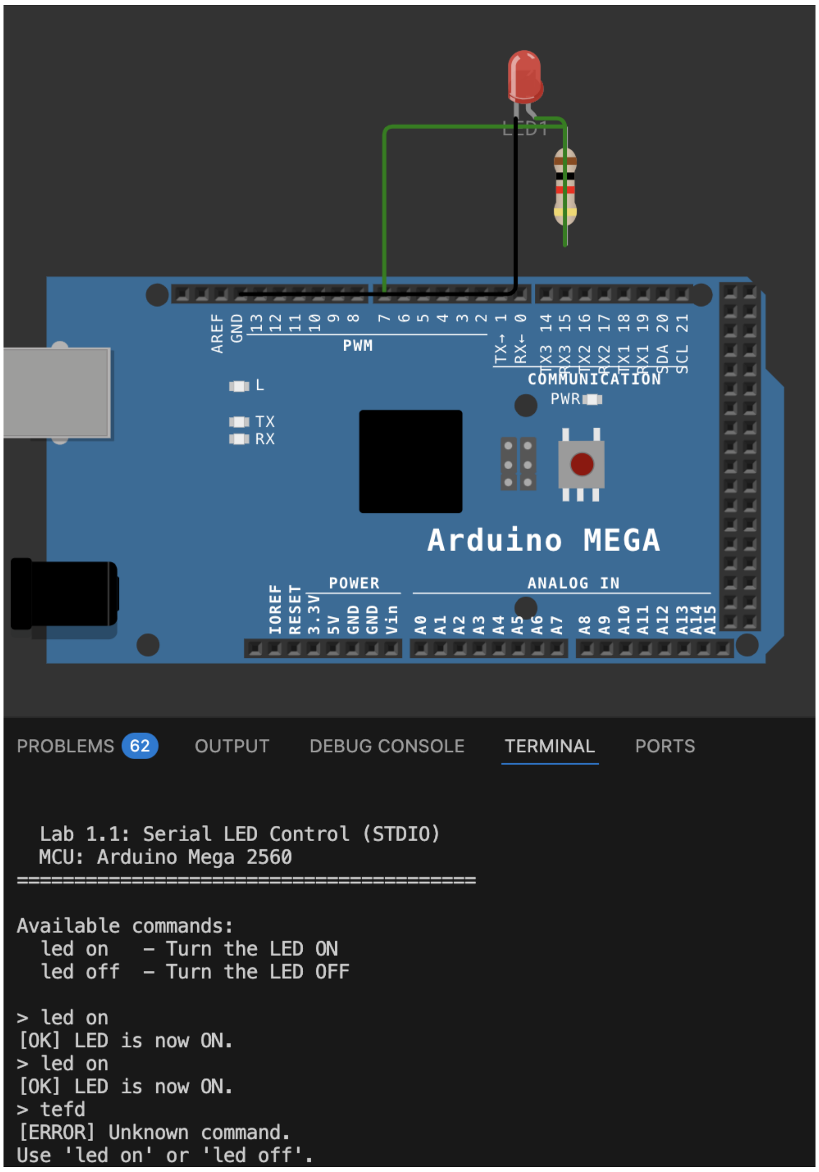
Test: Unknown Command Handling

Entering an unrecognized command triggers an error message:

***Listing 3.5** – Serial output — unknown command error*

```
> hello
[ERROR] Unknown command.
Use 'led on' or 'led off'.
>
```

The system does not crash or hang — it simply reports the error and returns to the prompt, ready for the next command.



```
  Lab 1.1: Serial LED Control (STDIO)
  MCU: Arduino Mega 2560
========================================

Available commands:
  led on   - Turn the LED ON
  led off  - Turn the LED OFF

> led on
[OK] LED is now ON.
> led on
[OK] LED is now ON.
> tefd
[ERROR] Unknown command.
Use 'led on' or 'led off'.
```

Figure 3.6 *– Error handling for unrecognized commands*

Test: Case Insensitivity

The command parser handles mixed-case input correctly:

Listing 3.6 *– Serial output — case insensitive commands*

```
> LED ON
[OK] LED is now ON.
> Led Off
[OK] LED is now OFF.
> LED on
[OK] LED is now ON.
>
```

This confirms that the `toLowerStr()` normalization in the parser works correctly.

Test: Leading and Trailing Whitespace

Commands with extra whitespace are handled gracefully:

Listing 3.7 *– Serial output — whitespace handling*

```
>    led on
[OK] LED is now ON.
> led off
[OK] LED is now OFF.
>
```

The `trimStr()` function strips leading and trailing whitespace before matching.

3.4. System Observations

The system demonstrates several important characteristics during operation. The LED state changes immediately upon pressing Enter with no perceptible delay, and the confirmation message appears within milliseconds, indicating responsive real-time behavior. The `fgets()` call blocks the main loop while waiting for input, which is acceptable for this simple application but would need to be replaced with non-blocking I/O in a multi-tasking system requiring concurrent operations.

Characters typed in the terminal are echoed back by the MCU (implemented in `serialGetChar()`), providing visual feedback to the user. The application uses minimal RAM, with the input buffer occupying 64 bytes and the parser using small stack-allocated buffers for processing. Finally, the Wokwi simulation accurately reproduces the serial communication and LED control behavior observed on physical hardware, validating the correctness of the implementation.

3.5. Verification Summary

- **Compilation:** Successful build with no errors or warnings.
- **Initialization:** STDIO serial redirection and LED initialization complete correctly. Welcome banner displays on startup.
- **LED ON command:** `led on` correctly turns the LED on with confirmation message.
- **LED OFF command:** `led off` correctly turns the LED off with confirmation message.
- **Error handling:** Unknown commands produce a clear error message without crashing.
- **Case insensitivity:** Commands work regardless of letter case.
- **Whitespace tolerance:** Leading/trailing whitespace is trimmed before parsing.
- **STDIO usage:** All text I/O uses `printf()` and `fgets()` (not `Serial.print()`).
- **Modularity:** Code is cleanly separated into reusable library modules.

4. Conclusion

4.1. Achievement of Objectives

All primary objectives of Laboratory Work 1.1 were successfully accomplished. The UART protocol was used to establish bidirectional text communication between the Arduino Mega 2560 and a host PC, demonstrating fundamental concepts of asynchronous serial communication including baud rate configuration, character transmission/reception, and echo handling. The C standard I/O library was successfully integrated into the embedded application through custom stream redirection, with `printf()` handling all text output (prompts, confirmations, error messages) and `fgets()` handling line-based input from the user. The AVR libc `fdev_setup_stream()` mechanism provided the bridge between portable C I/O and the hardware UART. A robust command parser was implemented that correctly interprets `led on` and `led off` commands with case-insensitive matching and whitespace tolerance, handling unknown commands gracefully with informative error messages. Finally, the application was decomposed into four independent modules (Led, StdioSerial, CommandParser, lab1_1_main), each with clear interfaces and separated concerns, making the library modules fully reusable in future laboratory work.

4.2. Performance Analysis and System Limitations

The system demonstrates strong performance characteristics. It responds to commands within milliseconds, well below human-perceptible latency. The memory footprint is minimal, with approximately 64 bytes for the input buffer and small stack allocations in the parser. The compiled firmware occupies only a small fraction of the ATmega2560's 256 KB Flash and 8 KB SRAM resources.

However, the system has several notable limitations. The most significant architectural limitation is the use of blocking I/O: the `fgets()` call blocks the main loop while waiting for serial input, preventing the MCU from performing other tasks during this time, such as reading sensors, updating displays, or running timers. The system only supports simple on/off commands, with no support for brightness control (PWM), blinking patterns, or status queries. Currently, only a single LED is controlled; extending to multiple LEDs would require parser modifications and additional LED instances. Additionally, the system has no persistent state—the LED state is lost on reset with no EEPROM storage for the last configuration. The baud rate (9600) is hardcoded and cannot be changed at runtime.

4.3. Proposed Improvements

Several enhancements could address the current limitations. Non-blocking I/O should replace the blocking `fgets()` approach by implementing character-by-character buffering in the main loop, allowing the MCU to service other peripherals between incoming characters. The command set could be extended to support additional commands such as `led toggle`, `led status`, `led blink` <rate>, or PWM-based `led brightness` <`0-255`>. The system could support multiple peripherals by extending the parser to control multiple LEDs, buttons, or other actuators by adding device identifiers to commands (e.g., `led1 on`, `led2 off`). Configuration persistence could be added by storing the last LED state in EEPROM to restore it after power cycling. A `help` command could be implemented to dynamically list all available commands and their descriptions. Finally, a simple command buffer could be added to allow the user to recall and re-execute previous commands.

4.4. Reflections on the Learning Experience

This laboratory work provided hands-on experience with several foundational concepts in embedded systems development. Understanding how the portable C I/O library can be adapted to work with microcontroller peripherals through custom stream implementations was a key insight, since this technique is widely used in professional embedded firmware for logging and debugging. Separating the application into distinct layers (driver, abstraction, service, application) required disciplined design thinking, and the resulting code is clean, testable, and reusable. Even a simple text-based command interface requires careful attention to user experience: echo, prompts, error messages, case handling, and whitespace tolerance all contribute to a usable system. Using Wokwi to test the application before physical deployment accelerated the development cycle and reduced the risk of hardware damage from wiring errors.

4.5. Impact of Technology in Real-World Applications

The serial command interface pattern demonstrated in this lab is directly applicable to numerous real-world embedded systems. Industrial automation systems such as SCADA and PLCs use serial protocols (RS-232, RS-485) for device configuration and monitoring, with text-based command structures similar to MODBUS ASCII or proprietary diagnostic interfaces. Many IoT and smart home devices provide serial debug consoles for firmware updates, Wi-Fi configuration, and troubleshooting; the ESP32, for example, uses a serial AT command interface for its Wi-Fi module. The OBD-II automotive standard includes serial-based diagnostic protocols for reading engine parameters and clearing fault codes. Medical devices employ serial interfaces for device calibration, data logging, and maintenance access in medical instrumentation. Mission-critical aerospace and defense systems use serial command interfaces for ground station communication and satellite command/telemetry links.

The modular software architecture employed in this lab (separation of drivers, services, and application logic) reflects industry best practices codified in standards such as AUTOSAR (AUTomotive Open System ARchitecture) for automotive embedded software.

5. Note Regarding Usage of AI

During the preparation of this laboratory report, the author utilized artificial intelligence tools to enhance the quality and presentation of the documentation. Specifically, the following AI tools were employed:

- **Report Formatting:** Assistance with LaTeX document structure, formatting consistency, and adherence to academic writing standards.
- **Report Template Generation:** Creation of the initial document template with proper sectioning and organizational structure.
- **Content Rephrasing:** Refinement of technical descriptions and explanations to ensure grammatical accuracy and clarity without altering the technical substance.
- **Code Documentation:** Generation of inline comments within the source code to improve code readability and maintainability.
- **Development Environment Setup:** Configuration assistance for PlatformIO integration with the Wokwi simulation platform.

All AI-generated content was thoroughly reviewed, validated, and adjusted by the author to ensure technical accuracy, alignment with project requirements, and adherence to embedded systems best practices. The core technical implementation, circuit design, system architecture decisions, and experimental results represent the author's original work and understanding of the subject matter. The AI tools served as assistive technologies to enhance documentation quality and development workflow efficiency, while the author retained full responsibility for the content, correctness, and conclusions presented in this report.

6. Bibliography

[1] Bragarenco, A., Astafi, V., *Sisteme Electronice Încorporate: Indicaţii metodice pentru lucrări de laborator*, Universitatea Tehnică a Moldovei, Chişinău, 2024.

[2] Atmel Corporation, *ATmega2560 Datasheet — 8-bit AVR Microcontroller with 256K Bytes In-System Programmable Flash*, Document 2549Q–AVR–02/2014, 2014.

[3] Arduino, *Arduino Mega 2560 Rev3 Documentation*, `https://docs.arduino.cc/hardware/uno-rev3/`, Accessed: 2026-02-18.

[4] AVR Libc Reference Manual, *Standard I/O Facilities — <stdio.h>*, `https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html`, Accessed: 2026-02-18.

[5] PlatformIO, *Professional Collaborative Platform for Embedded Development*, `https://platformio.org/`, Accessed: 2026-02-18.

[6] Wokwi, *Online Electronics Simulator — Arduino, ESP32, Raspberry Pi Pico*, `https://wokwi.com/`, Accessed: 2026-02-18.

7. Appendix

The complete source code for Laboratory Work 1.1 is organized below by architectural layer. The full project is available in the GitHub repository: `https://github.com/mcittkmims/es-labs`.

7.1. Application Layer

Entry Point: main.cpp

Listing 7.1 – src/main.cpp — Application entry point (lab selector)

```cpp
/**
 * @file main.cpp
 * @brief Application Entry Point - Lab Selector
 */

#include <Arduino.h>

// Select the active lab
#include "lab1_1_main.h"

void setup() {
    lab1_1Setup();
}

void loop() {
    lab1_1Loop();
}
```

Lab 1.1 Main Module

Listing 7.2 – lab/lab1_1/lab1_1_main.h — Lab 1.1 interface

```cpp
#ifndef LAB1_1_MAIN_H
#define LAB1_1_MAIN_H

void lab1_1Setup();
void lab1_1Loop();

#endif // LAB1_1_MAIN_H
```

Listing 7.3 – lab/lab1_1/lab1_1_main.cpp — Lab 1.1 implementation

```cpp
#include "lab1_1_main.h"
#include <Arduino.h>
#include <stdio.h>
```

```
#include "Led.h"
#include "StdioSerial.h"
#include "CommandParser.h"

// Pin Configuration
static const uint8_t LED_PIN = 7;
static const unsigned long BAUD_RATE = 9600;

// Module-level objects
static Led led(LED_PIN);
static char inputBuffer[64];

void lab1_1Setup() {
    stdioSerialInit(BAUD_RATE);
    led.init();

    printf("\r\n");
    printf("========================================\r\n");
    printf("  Lab 1.1: Serial LED Control (STDIO)\r\n");
    printf("  MCU: Arduino Mega 2560\r\n");
    printf("========================================\r\n");
    printf("\r\n");
    printf("Available commands:\r\n");
    printf("  led on   - Turn the LED ON\r\n");
    printf("  led off  - Turn the LED OFF\r\n");
    printf("\r\n");
}

void lab1_1Loop() {
    printf("> ");

    if (fgets(inputBuffer, sizeof(inputBuffer), stdin) != NULL) {
        CommandType cmd = parseCommand(inputBuffer);

        switch (cmd) {
            case CMD_LED_ON:
                led.turnOn();
                printf("[OK] LED is now ON.\r\n");
                break;
            case CMD_LED_OFF:
                led.turnOff();
                printf("[OK] LED is now OFF.\r\n");
                break;
            case CMD_UNKNOWN:
                printf("[ERROR] Unknown command.\r\n");
                printf("Use 'led on' or 'led off'.\r\n");
                break;
        }
    }
}
```

7.2. Service Layer

CommandParser Module

Listing 7.4 – lib/CommandParser/CommandParser.h — Command parser interface

```
#ifndef COMMAND_PARSER_H
#define COMMAND_PARSER_H

#include <stdint.h>

enum CommandType {
    CMD_UNKNOWN,
    CMD_LED_ON,
    CMD_LED_OFF
};

CommandType parseCommand(const char *input);

#endif // COMMAND_PARSER_H
```

Listing 7.5 – lib/CommandParser/CommandParser.cpp — Command parser implementation

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

static const uint8_t PARSE_BUFFER_SIZE = 64;

static void toLowerStr(char *dest, const char *src, size_t maxLen) {
    size_t i = 0;
    while (src[i] != '\0' && i < maxLen - 1) {
        dest[i] = tolower((unsigned char)src[i]);
        i++;
    }
    dest[i] = '\0';
}

static void trimStr(char *dest, const char *src, size_t maxLen) {
    while (*src && isspace((unsigned char)*src)) {
        src++;
    }
    size_t len = strlen(src);
    while (len > 0 && isspace((unsigned char)src[len - 1])) {
        len--;
    }
    if (len >= maxLen) {
        len = maxLen - 1;
    }
    strncpy(dest, src, len);
    dest[len] = '\0';
}

CommandType parseCommand(const char *input) {
    char trimmed[PARSE_BUFFER_SIZE];
    char lower[PARSE_BUFFER_SIZE];

    trimStr(trimmed, input, sizeof(trimmed));
    toLowerStr(lower, trimmed, sizeof(lower));

    if (strcmp(lower, "led on") == 0) {
        return CMD_LED_ON;
    } else if (strcmp(lower, "led off") == 0) {
        return CMD_LED_OFF;
    }

    return CMD_UNKNOWN;
}
```

7.3. ECU Abstraction Layer

StdioSerial Module

Listing 7.6 *– lib/StdioSerial/StdioSerial.h — STDIO redirection interface*

```
#ifndef STDIO_SERIAL_H
#define STDIO_SERIAL_H

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

void stdioSerialInit(unsigned long baudRate);

#endif // STDIO_SERIAL_H
```

Listing 7.7 *– lib/StdioSerial/StdioSerial.cpp — STDIO redirection implementation*

```
#include "StdioSerial.h"

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

```
static int serialGetChar(FILE *stream) {
    while (!Serial.available()) {
        // Wait for input
    }
    char c = Serial.read();

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

static FILE serialStream;

void stdioSerialInit(unsigned long baudRate) {
    Serial.begin(baudRate);
    while (!Serial) { ; }
    fdev_setup_stream(&serialStream,
        serialPutChar, serialGetChar, _FDEV_SETUP_RW);
    stdout = &serialStream;
    stdin  = &serialStream;
}
```

7.4. Microcontroller Abstraction Layer

Led Driver Module

***Listing 7.8** – lib/Led/Led.h — LED driver interface*

```
#ifndef LED_H
#define LED_H

#include <Arduino.h>

class Led {
public:
    Led(uint8_t pin);
    void init();
    void turnOn();
    void turnOff();
    void toggle();
    bool isOn() const;
private:
    uint8_t ledPin;
    bool state;
};

#endif // LED_H
```

***Listing 7.9** – lib/Led/Led.cpp — LED driver implementation*

```
#include "Led.h"

Led::Led(uint8_t pin) : ledPin(pin), state(false) {}

void Led::init() {
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
    state = false;
}

void Led::turnOn() {
    digitalWrite(ledPin, HIGH);
    state = true;
}

void Led::turnOff() {
    digitalWrite(ledPin, LOW);
    state = false;
}

void Led::toggle() {
    if (state) {
        turnOff();
    } else {
        turnOn();
    }
}

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

7.5. Configuration Files

Listing 7.10 *– platformio.ini — PlatformIO project configuration*

```
[env:lab1_1]
platform = atmelavr
board = megaatmega2560
framework = arduino
monitor_speed = 9600
build_src_filter = +<*> +<../lab/lab1_1/*>
build_flags = -I lab/lab1_1
```