MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

DANIELA COJOCARI, FAF-231

Report

Laboratory work n.1

of Embedded Systems

User Interaction: STDIO – Serial Interface

Checked by:
Alexei Martiniuc

Chișinău, 2026

1. Analysis of the Situation in the Field

1.1 Description of the technologies used and the context of the developed application.

STDIO Library – The stdio.h header provides generic file operation support and supplies functions with narrow character input/output capabilities. I/O streams are denoted by objects of type FILE that can only be accessed and manipulated through pointers of type FILE*. Each stream is associated with an external physical device (file, standard input stream, printer, serial port, etc) [1].

Serial Interface – A serial interface is a communication interface between two digital systems that transmits data as a series of voltage pulses down a wire. A ”1” is represented by a high logical voltage and a ”0” is represented by a low logical voltage. Essentially, the serial interface encodes the bits of a binary number by their ”temporal” location on a wire rather than their ”spatial” location within a set of wires. Encoding data bits by their ”spatial” location is referred to as a parallel interface and encoding bits by their ”temporal” location is referred to as a serial interface [2].

PlatformIO – PlatformIO is a cross-platform, cross-architecture, multiple framework, professional tool for embedded systems engineers and for software developers who write applications for embedded products. The build system structure automatically tags software dependencies and applies them using a modular hierarchy that takes away the usual complexity and pain [3].

Wokwi Simulator – Wokwi is an embedded systems and IoT simulator supporting ESP32, Arduino, and the Raspberry Pi Pico [4]. It provides a safe environment where mistakes don’t damage hardware, so you can experiment and debug confidently. Projects are easy to share and collaborate on, and you have access to unlimited virtual components without worrying about cost or stock. Its unique features include WiFi simulation, a virtual logic analyzer for capturing digital signals, advanced debugging with GDB, SD card simulation, a Chips API for creating custom components, and integration with Visual Studio Code, making it a comprehensive tool for learning, prototyping, and testing embedded systems.

Microcontroller – A microcontroller unit (MCU) is essentially a small computer on a single chip. It is designed to manage specific tasks within an embedded system without requiring a complex operating system. These compact integrated circuits (ICs) contain a processor core (or cores), random-access memory (RAM) and electrically erasable programmable read-only memory (EEPROM) for storing the custom programs that run on the microcontroller, even when the unit is disconnected from a power supply [5].

Arduino – Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - light on a sensor, a finger on a button, or a Twitter message -

and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing [6].

The application is developed in the context of embedded systems education, aiming to familiarize students with serial communication using the STDIO library and with designing simple, modular microcontroller-based solutions. The project focuses on creating an application that receives text commands from a serial terminal to control a peripheral device, specifically an LED, providing immediate textual feedback to confirm command execution. The system is implemented on a microcontroller such as Arduino Uno, with hardware components including LEDs, resistors, and a breadboard for prototyping. Development is supported by PlatformIO in Visual Studio Code, with optional use of simulators like Wokwi or Proteus for testing. By separating functionality for each peripheral into modular files and adhering to coding conventions, the project not only demonstrates the principles of serial I/O and command interpretation but also encourages reusable design and rapid, safe experimentation within a virtual or real embedded environment.

1.2 Presentation of the hardware and software components used, explaining the role of each.

Arduino Uno – The Arduino Uno is a widely used microcontroller development board based on the ATmega328P chip. It features digital and analog I/O pins that allow it to interface with sensors, LEDs, and other peripherals. The board can be powered and programmed through a USB connection and supports serial communication, which is essential for receiving commands from a terminal in this project. It acts as the central "brain" of the system, executing the code that interprets serial commands and controls external components like LEDs.

LEDs – Light-Emitting Diodes (LEDs) are simple semiconductor devices that emit light when an electric current passes through them. They are used in this project as visual indicators to show the result of serial commands (e.g., LED ON or LED OFF). Because LEDs require only small currents, they are suitable for direct control by microcontroller pins.

220 Ω resistor – The 220 Ω resistor is placed in series with the LED to limit the current flowing through it, preventing damage to both the LED and the microcontroller. Without a resistor, the LED could draw excessive current, potentially leading to failure of the component or the microcontroller pin. Choosing an appropriate resistor value ensures the LED operates within safe current limits.

Breadboard – A breadboard is a solderless prototyping board that allows components and wires to be

connected easily without permanent soldering. It provides a flexible way to build and test circuits, making it ideal for educational projects and rapid experimentation. The Arduino, resistors, LEDs, and jumper wires are all placed onto the breadboard to create the necessary electrical connections for the circuit.

Connection cables (jumper wires) – Jumper wires are used to make electrical connections between the Arduino board, the breadboard, and the various components in the circuit. They come with connector pins that fit into the board headers or breadboard holes, enabling modular and reconfigurable wiring. They are essential for linking the microcontroller's pins to the LED and resistor circuits.

Power supply (USB) – The USB power supply provides both power and data connectivity between the Arduino Uno and a computer. It supplies the 5 V needed to power the microcontroller and peripherals while also enabling code uploads and serial communication with the terminal. Using USB simplifies both development and testing, as the board can run directly from the computer without needing a separate external power source.

Visual Studio Code with PlatformIO extension – Visual Studio Code (VS Code) is a versatile code editor, and when combined with the PlatformIO extension, it becomes a powerful environment for embedded systems development. PlatformIO supports multiple microcontroller platforms and frameworks and handles project configuration, compilation, and uploading of firmware to the Arduino. It simplifies dependency management and allows advanced features like unit testing and integrated terminal access. This environment is used to write, build, and deploy the application that reads serial input and controls the LED.

Serial terminal emulator (e.g., PlatformIO Serial Monitor, TeraTerm, PuTTY) – A serial terminal emulator is software that connects to the Arduino's serial port and allows text communication with the microcontroller. In this project, the terminal is used to send commands such as "led on" and "led off" to the Arduino and to view textual feedback confirming command reception. The emulator displays the serial output and serves as the user interface for testing serial communication functionality.

Hardware simulator - Wokwi – Wokwi is an embedded systems and IoT simulator that supports Arduino, ESP32, Raspberry Pi Pico, and other microcontroller platforms. It can be used directly in a browser or integrated into VS Code. Wokwi enables simulation of embedded circuits and serial communication without physical components, allowing developers to prototype and debug designs virtually. It offers features such as WiFi simulation, logic analyzers, and interactive virtual components, making it a convenient tool for experimentation and verification before deploying to real hardware.

1.3 Explanations of the system architecture and justification for the chosen solution.

The developed system follows a modular embedded architecture in which responsibilities are clearly separated into independent components. The application layer manages command interpretation and user interaction, the service layer handles serial communication through the STDIO library, and the device driver layer controls the hardware peripheral (LED). This layered structure ensures that each module has a single, well-defined role, improving clarity and maintainability.

The serial communication is implemented by redirecting the standard input and output streams (stdin, stdout, stderr) to the Arduino serial interface using fdevopen(). This allows the use of standard C functions such as printf() and fgets() instead of Arduino-specific functions. As a result, the application benefits from portability, cleaner syntax, and better abstraction of the communication mechanism. The service layer acts as an intermediary between the hardware UART interface and the application logic, ensuring proper separation between low-level data transmission and high-level command processing.

The LED functionality is encapsulated in a dedicated device driver module, which manages pin configuration and digital output control. By isolating hardware access within this driver, the application layer remains independent of direct hardware manipulation. This approach supports modular development and allows the LED driver to be reused or extended in future projects involving additional peripherals.

The chosen solution aligns with the educational objectives of understanding serial communication principles and using the STDIO library for text-based data exchange. Moreover, the architecture supports scalability, reusability, and compatibility with both real hardware and simulation environments such as Wokwi. Overall, the design reflects good embedded systems engineering practices by promoting abstraction, modularity, and clear separation between hardware and software components.

1.4 A relevant case study demonstrating the applicability of the proposed solution.

The architecture implemented in this project reflects real-world embedded system designs commonly used in industrial equipment, IoT devices, and diagnostic systems. Serial text-based command interfaces are widely employed in embedded products for configuration, testing, and maintenance purposes. For example, many network routers, industrial controllers, and development boards provide a serial console interface that allows engineers to send textual commands to configure parameters, enable or disable peripherals, or retrieve system status information. The mechanism implemented in this laboratory, interpreting commands such as “led on” and “led off” and providing textual confirmation, is conceptually identical to command-line interfaces used in professional embedded systems.

In industrial automation, microcontroller-based control units frequently use serial communication (UART)

to interact with supervisory systems or human-machine interfaces (HMI). Text-based protocols are often chosen for simplicity, readability, and ease of debugging. Similarly, in IoT development and prototyping, serial terminals are used during firmware development to monitor logs, send configuration commands, and validate hardware behavior. The modular separation between communication services and hardware drivers, as implemented in this project, mirrors the architecture of scalable embedded firmware used in commercial products.

Therefore, the proposed solution is directly applicable to real embedded environments such as device configuration consoles, bootloader interfaces, smart home controllers, industrial monitoring systems, and debugging interfaces for microcontroller-based products. The use of STDIO redirection and layered architecture demonstrates principles that are fundamental in professional embedded software engineering.

2. Implementation

2.1 Architectural Sketch

The architectural sketch illustrates a layered embedded system structure designed to clearly separate software responsibilities from hardware implementation. This approach follows well-established embedded systems engineering principles, where abstraction layers isolate high-level application logic from low-level hardware control. The diagram presents a top-down interaction model, starting from the Application layer and progressing through multiple software abstraction layers down to the physical hardware components, including the microcontroller and the LED.

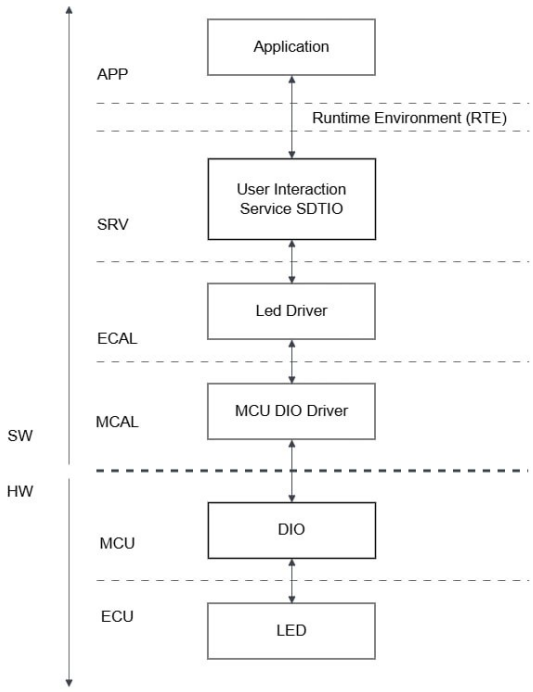

Figure 0.0.1 - Architectural Sketch

Application (APP Layer)

The Application layer represents the highest level of the software architecture and contains the main program logic. In this laboratory project, this layer is responsible for interpreting user commands received through the serial interface, such as “led on” and “led off.” It makes decisions based on input and triggers appropriate actions by calling lower-level services. Importantly, the application does not directly access hardware registers or manipulate pins; instead, it relies on abstraction layers below it, ensuring independence from hardware-specific details.

Runtime Environment (RTE)

The Runtime Environment acts as an intermediary between the Application layer and the lower-level services. It provides standardized interfaces that allow the application to interact with communication and hardware services without knowing their internal implementation. This layer ensures portability and flexibility, meaning that underlying drivers can be modified or replaced without affecting the application logic. The RTE therefore plays a key role in decoupling business logic from hardware dependencies.

User Interaction Service – STDIO (SRV Layer)

The User Interaction Service belongs to the Service (SRV) layer and is responsible for handling serial communication using the STDIO abstraction. It redirects standard input and output streams (stdin, stdout, stderr) to the serial interface, allowing the use of standard C functions such as ‘printf()‘ and ‘fgets()‘. This greatly simplifies communication by abstracting the UART hardware details and presenting them as standard streams. In this laboratory, it enables text-based interaction between the user and the microcontroller through a serial terminal.

LED Driver (ECAL Layer)

The LED Driver is part of the ECU Abstraction Layer (ECAL) and is responsible for controlling the LED peripheral. It encapsulates functions such as initializing the LED pin and setting its state to HIGH or LOW. By isolating LED-specific logic in a dedicated driver, the system ensures that the application layer does not directly manipulate hardware pins. This modular approach allows the LED driver to be reused or extended for additional peripherals in future projects.

MCU DIO Driver (MCAL Layer)

The MCU DIO (Digital Input/Output) Driver belongs to the Microcontroller Abstraction Layer (MCAL). This layer provides low-level control over the microcontroller’s digital I/O registers. It is responsible for configuring pin directions and writing logical values to output pins. The MCAL layer interacts directly with microcontroller hardware registers, translating high-level function calls into actual electrical signals. It forms the bridge between software logic and the microcontroller’s internal peripherals.

DIO (MCU Hardware Layer)

The DIO block represents the digital input/output hardware module inside the microcontroller. This

hardware peripheral physically controls the voltage levels on the microcontroller pins. When instructed by the MCAL driver, the DIO module sets a pin to HIGH (typically 5V) or LOW (0V), thereby enabling or disabling current flow to connected components. It is the first hardware layer below the software stack.

LED (ECU Layer)

The LED is the final physical component in the architecture and represents the external electronic control unit (ECU) element in this simplified system. It acts as a visual output device that responds to voltage changes on the microcontroller pin. When the pin is set HIGH, current flows through the resistor and LED, causing it to emit light; when set LOW, the current stops and the LED turns off. The LED therefore provides tangible, real-world feedback that confirms correct operation of the entire software and hardware stack.

Overall, this layered architecture clearly demonstrates how user commands propagate through successive abstraction layers, from high-level application logic down to physical hardware, ensuring clean separation of concerns and structured embedded system design.

2.2 Functional Block Diagrams

Flowcharts are a fundamental tool in system design and software development, providing a visual representation of the logical flow of a program or process. In the context of this application, flowcharts help illustrate how the system interprets serial commands, processes user inputs, and controls hardware components like LEDs.

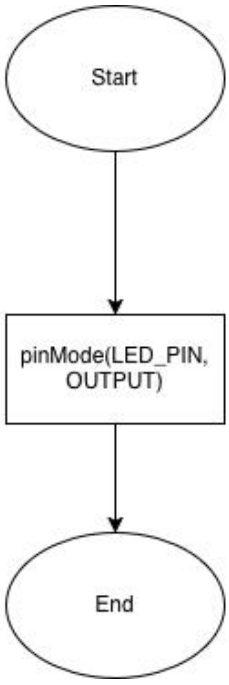

Figure 0.0.2 - ddLedSetup()

The ddLedSetup() function [0.0.2] is responsible for initializing the LED pin as an output. By calling

pinMode(LED_PIN, OUTPUT), it configures the microcontroller so that current can flow through the pin to drive the LED. This initialization step is essential in embedded systems, as it ensures that the hardware is prepared before any control commands are executed.

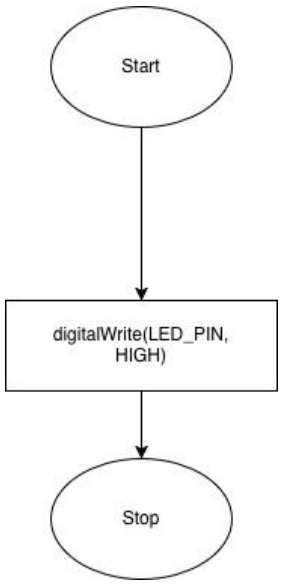

Figure 0.0.3 - ddLedTurnOn()

The ddLedTurnOn() function [0.0.3] sets the LED pin to a HIGH state using digitalWrite(LED_PIN, HIGH), which allows current to flow through the LED and resistor, causing it to illuminate. This function provides a direct example of digital output control, showing how software commands translate immediately into physical effects on hardware.

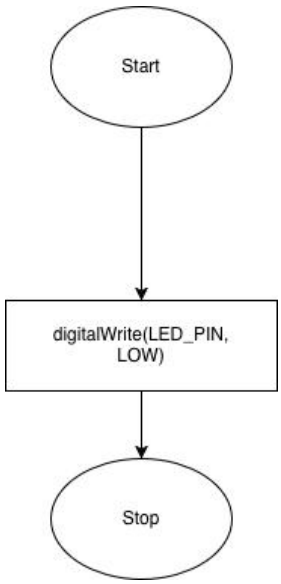

Figure 0.0.4 - ddLedTurnOff()

The ddLedTurnOff() function [0.0.4] switches the LED pin to a LOW state by executing digitalWrite(LED_PIN LOW), stopping current flow and turning the LED off. Similar to ddLedTurnOn(), it is a single-step function with a simple flowchart representation.

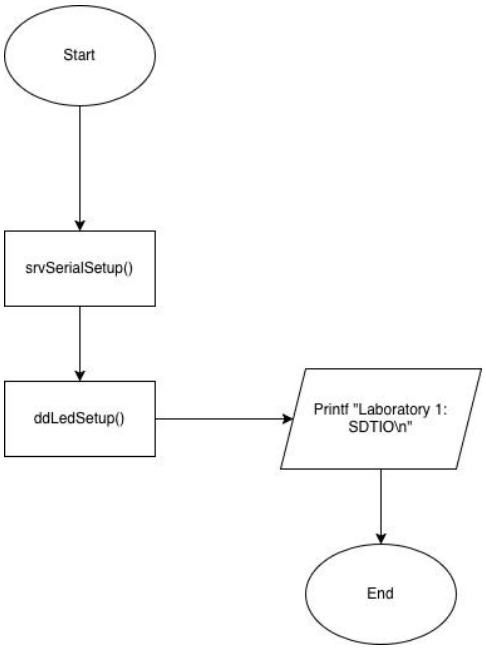

Figure 0.0.5 - lab1Setup()

The lab1Setup() function [0.0.5] is responsible for initializing the laboratory environment at the start of the program. It first calls srvSerialSetup() to configure the serial communication driver, enabling the microcontroller to send and receive textual data through the serial terminal. Next, it calls ddLedSetup() to configure the LED pin as an output, ensuring the hardware is ready for control. Finally, it prints a welcome message to the terminal, confirming that the setup process has completed. This function represents the initialization phase of the application, where both software services and hardware peripherals are prepared before entering the main execution loop.

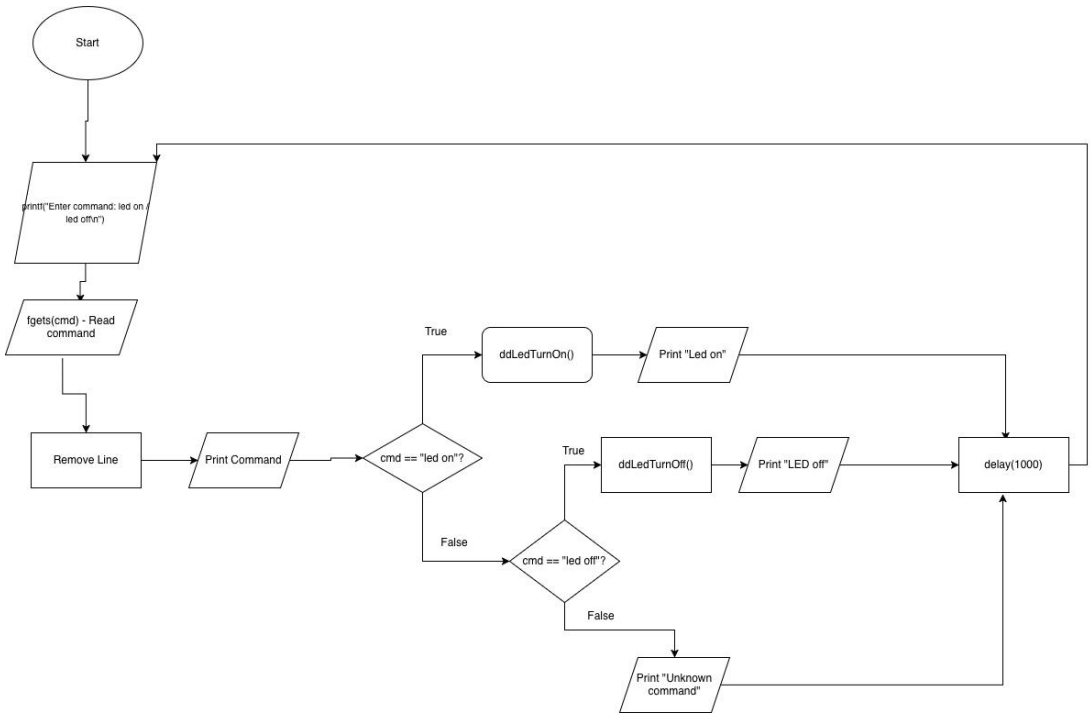

Figure 0.0.6 - lab1Loop()

The lab1Loop() function [0.0.6] contains the main execution loop of the application and handles user interaction through the serial terminal. It first prompts the user to enter a command using printf(), then reads the input with fgets() and removes any trailing newline character to sanitize the command. The function then displays the received command and processes it using conditional statements: if the command is ”led on”, it calls ddLedTurnOn() and confirms the action; if ”led off”, it calls ddLedTurnOff() and prints a confirmation. Any unrecognized command triggers an ”Unknown command” message. Finally, the function waits one second using delay(1000) before repeating. This function demonstrates real-time command processing, text-based input parsing, and modular hardware control.

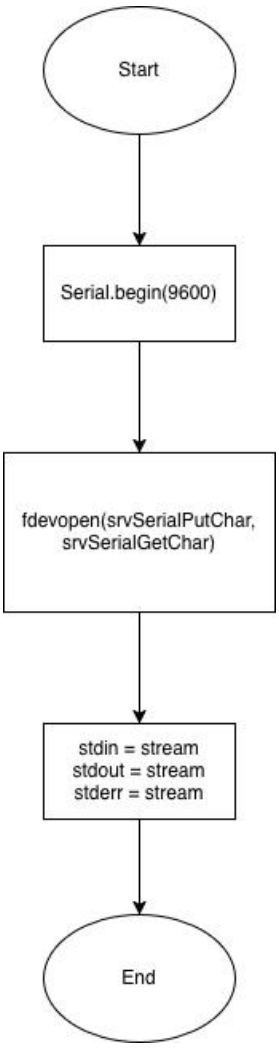

Figure 0.0.7 - srvSerialSetup()

The srvSerialSetup() function [0.0.7] initializes the serial communication driver for the microcontroller. It begins by calling Serial.begin(9600) to configure the hardware UART for communication at a baud rate of 9600 bps. Then, it creates a standard C FILE stream using fdevopen(), linking the srvSerialPutChar and srvSerialGetChar functions to handle output and input, respectively. Finally, it redirects the standard input (stdin), output (stdout), and error (stderr) streams to this serial stream. This setup allows higher-level code to use standard C I/O functions like printf() and fgets() for serial communication, abstracting the hardware specifics.

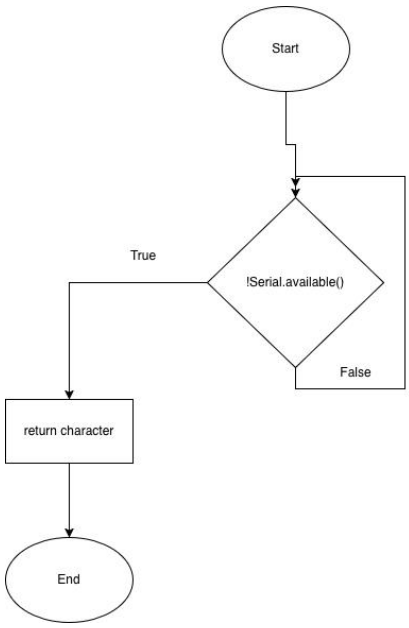

Figure 0.0.8 - srvSerialGetChar()

The srvSerialGetChar() function [0.0.8] reads a single character from the serial interface. It waits until data is available on the UART using a blocking loop (while (!Serial.available());) and then returns the character with Serial.read().

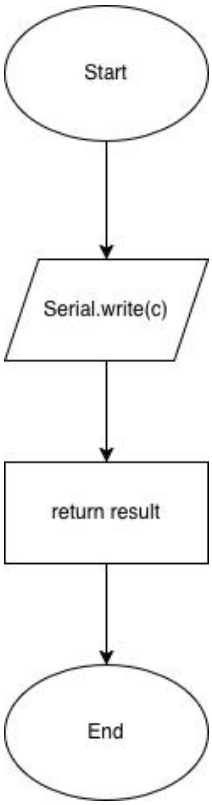

Figure 0.0.9 - srvSerialPutChar()

The srvSerialPutChar() function [0.0.9] writes a single character to the serial interface using Serial.write(c).

It is called whenever a higher-level function such as printf() outputs text to the serial terminal.

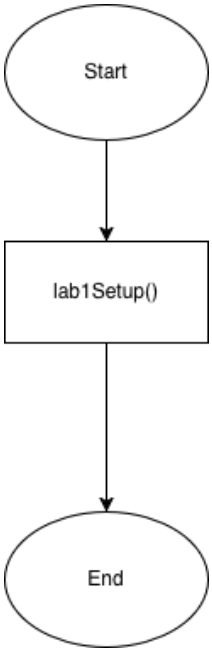

Figure 0.0.10 - setup()

The setup() function [0.0.10] in Arduino is executed once when the microcontroller starts or resets. In this program, it serves as the entry point for the selected laboratory application. The preprocessor directive #if APP_NAME == LAB_1 ensures that lab1Setup() is called only when the LAB_1 application is selected, allowing the same framework to support multiple laboratory exercises with minimal changes.

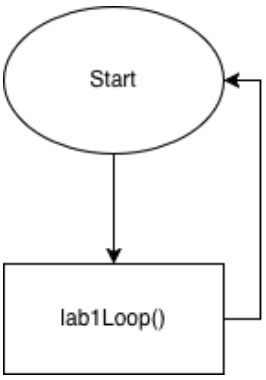

Figure 0.0.11 - loop()

The loop() function [0.0.11] in Arduino runs continuously after setup() completes, forming the main execution cycle of the microcontroller. Here, it conditionally calls lab1Loop() when APP_NAME is set to LAB_1, ensuring that the main laboratory logic executes repeatedly.

2.3 Electrical Sketch

The electrical sketch illustrates a simple LED control circuit using an Arduino Uno as the central microcontroller. The Arduino acts as the brain of the system, receiving commands from a serial terminal and controlling the LED accordingly.

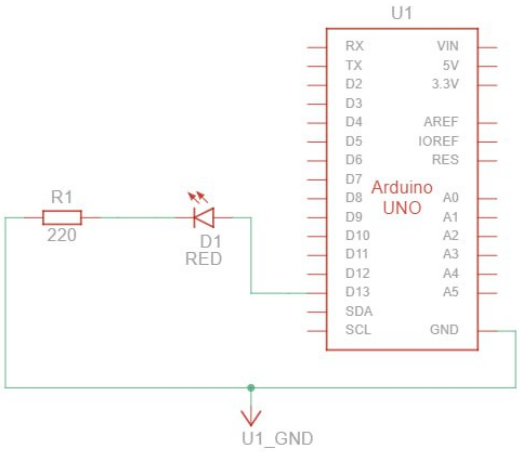

Figure 0.0.12 - Electrical Sketch

In this schematic, digital pin D13 is used as the output pin to drive the LED, while the GND pin provides a common reference for completing the circuit. This setup is typical for basic embedded systems projects, where one microcontroller pin controls a single output device.

The LED in the circuit serves as a visual indicator of command execution. Its anode is connected to a 220 Ω current-limiting resistor, which is in turn connected to the Arduino output pin. The cathode of the LED is connected to the Arduino ground. The resistor is crucial because it ensures that the current flowing through the LED remains within safe limits, preventing damage to both the LED and the microcontroller pin. Without this resistor, the LED could draw excessive current, potentially causing hardware failure.

Functionally, when the Arduino sets pin D13 HIGH, current flows through the resistor and LED to ground, illuminating the LED. Conversely, when the pin is set LOW, the current stops, and the LED turns off. This simple series connection demonstrates the basic principles of digital output control in embedded systems. It also provides a tangible way to verify that the serial commands sent from a terminal are correctly interpreted by the microcontroller.

Overall, this electrical sketch represents a minimal and safe experimental setup for testing the application's functionality. It is compatible with both physical prototyping on a breadboard and virtual simulation environments like Wokwi, allowing students to experiment without risk of damaging hardware.

3. Results

The first photo [0.0.13] illustrates the initial state of the LED, showing that it is off before any command is sent. This confirms that the system starts in a known default state and that the LED driver initialization (*ddLedSetup()*) does not inadvertently activate the hardware. It also demonstrates that the serial interface is ready to accept commands without affecting the peripheral's state.

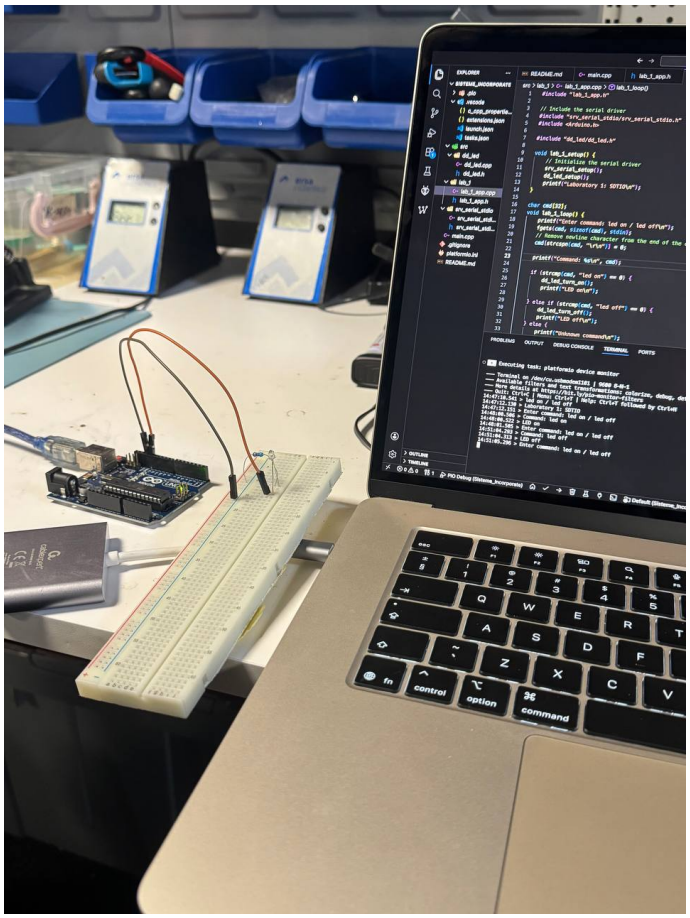

Figure 0.0.13 - Demonstration of Led Off

The second photo [0.0.14] shows the LED turned on after the "led on" command is sent from the serial terminal. This validates the full command-processing pipeline: the input is read through the serial driver (*srvSerialGetChar()*), processed in the application loop (*lab1Loop()*), and executed via the LED driver (*ddLedTurnOn()*). The visual change in the LED provides immediate feedback, confirming that the system correctly interprets user commands and controls the hardware as intended.

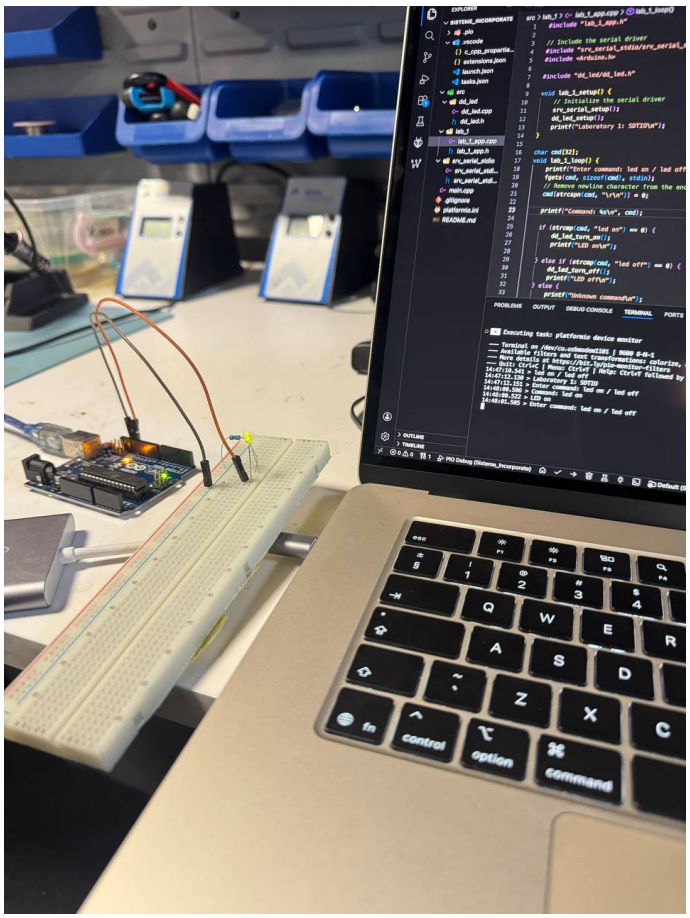

Figure 0.0.14 - Demonstration of Led On

Together, these two images provide a clear before-and-after demonstration of the system's functionality, highlighting the interactive control of hardware via serial commands and the successful implementation of the layered, modular architecture.

4. Conclusion

The laboratory project successfully demonstrates the principles of serial communication and modular embedded system design through a fully physical implementation using an Arduino Uno, LED, resistor, and breadboard. By employing a layered architecture, including application logic, service (STDIO) drivers, and hardware abstraction via device drivers, the system clearly separates responsibilities, promoting clarity, reusability, and maintainability. The design allows the application to process textual commands from a serial terminal, interpret them, and control the LED in real time, providing a direct and tangible demonstration of the input-processing-output cycle in embedded systems.

The physical implementation confirmed the functionality of each software module and its interaction with hardware: initialization routines properly configured the LED and serial interface, command handling reliably activated or deactivated the LED, and feedback messages displayed on the terminal confirmed successful operation. This approach emphasizes best practices in embedded development, such as modular coding, abstraction of hardware, and structured flow of control.

Overall, the project not only validates the correctness of the software and hardware integration but also reinforces key concepts in microcontroller-based system design, serial I/O communication, and peripheral control. The hands-on experience gained from physically building and testing the circuit provides a solid foundation for tackling more complex applications in real-world embedded and IoT systems.

5. Note Regarding the Use of AI Tools

During the preparation of this report, the author used ChatGPT to help structure points and organize content. All information generated with the tool was reviewed, validated, and adjusted to meet the requirements of the laboratory work.

Bibliography

[1] File input/output.
https://en.cppreference.com/w/c/io.html

[2] What is a serial interface?
https://academicweb.nd.edu/~lemmon/courses/ee224/web-manual/web-manual/lab9/node4.html

[3] What is PlatformIO?
https://docs.platformio.org/en/latest/what-is-platformio.html

[4] Welcome to Wokwi!
https://docs.wokwi.com/

[5] What is a microcontroller?
https://www.ibm.com/think/topics/microcontroller

[6] What is Arduino?
https://docs.arduino.cc/learn/starting-guide/whats-arduino/

Appendix - Source Code

dd_led

```
//dd_led.h


#ifndef DD_LED_H
#define DD_LED_H


#include <Arduino.h>


#ifndef LED_BUILTIN
#define LED_BUILTIN 13
#endif


#define LED_PIN LED_BUILTIN


void ddLedSetup();
void ddLedTurnOn();
void ddLedTurnOff();


#endif //DD_LED_H
```

```
//dd_led.cpp


#include "dd_led.h"


void ddLedSetup() {
    pinMode(LED_PIN, OUTPUT);
}


void ddLedTurnOn() {
    digitalWrite(LED_PIN, HIGH);
}


void ddLedTurnOff() {
```

```
    digitalWrite(LED_PIN, LOW);
}
```

srv_serial_stdio

```
//srv_serial_stdio.h

#ifndef SRV_SERIAL_STDIO_H
#define SRV_SERIAL_STDIO_H

#include <stdio.h>

void srvSerialSetup();
int srvSerialGetChar(FILE *f);
int srvSerialPutChar(char c, FILE *f);

#endif //SRV_SERIAL_STDIO_H
```

```
//srv_serial_stdio.cpp

#include "srv_serial_stdio.h"

// Include device driver
#include <Arduino.h>
#include <stdio.h>



// Initialize driver
void srvSerialSetup() {
    Serial.begin(9600);

    // Create a stream that uses the serial driver
    FILE *srv_serial_stdio_stream = fdevopen(srvSerialPutChar, srvSerialGetChar);

    stdin = srv_serial_stdio_stream;
```

```
    stdout = srv_serial_stdio_stream;
    stderr = srv_serial_stdio_stream;
}


// Define the function to get a character from the serial driver
int srvSerialGetChar(FILE *f) {
    // Wait for data to be available
    while (!Serial.available());
    return Serial.read();
}


// Define the function to put a character to the serial driver
int srvSerialPutChar(char c, FILE *f) {
    // Write the character to the serial driver
    return Serial.write(c);
}
```

lab_1_app

```
//lab_1_app.h


#ifndef LAB_1_APP_H
#define LAB_1_APP_H


void lab1Setup();
void lab1Loop();



#endif //LAB_1_APP_H
```

```
//lab_1_app.cpp


#include "lab_1_app.h"
```

```
// Include the serial driver
#include "srv_serial_stdio/srv_serial_stdio.h"
#include <Arduino.h>

#include "dd_led/dd_led.h"

void lab1Setup() {
    // Initialize the serial driver
    srvSerialSetup();

    // Initialize the LED driver
    ddLedSetup();
    printf("Laboratory 1: SDTIO\n");
}

char cmd[32];
void lab1Loop() {
    printf("Enter command: led on / led off\n");
    fgets(cmd, sizeof(cmd), stdin);
    // Remove newline character from the end of the command
    cmd[strcspn(cmd, "\r\n")] = 0;

    printf("Command: %s\n", cmd);

    // Process the command
    if (strcmp(cmd, "led on") == 0) {
        ddLedTurnOn();
        printf("LED on\n");

    } else if (strcmp(cmd, "led off") == 0) {
        ddLedTurnOff();
        printf("LED off\n");
    } else {
```

```
        printf("Unknown command\n");
    }

    delay(1000);
}
```

main

```
\\main.cpp

#include <Arduino.h>

#include "lab_1/lab_1_app.h"

#define APP_NAME LAB_1

void setup() {
    #if APP_NAME == LAB_1
        lab1Setup();
    #endif
}

void loop() {
    #if APP_NAME == LAB_1
        lab1Loop();
    #endif
}
```

Github link: https://github.com/DannaCojocari/Sisteme_Incorporate