Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Embedded Systems

Laboratory work work #1.1

User interaction via Serial Communication. Led

During the preparation of this report, the author used various AI Tools such as ChatGPT, Gemini, Claude to generate/augment the content, generate the code and structure the directory. The resulting information was reviewed, validated, and adjusted to meet the requirements of the laboratory assignment.

Author: Timur Cravtov
std. gr. FAF-231

Verified by:
Alexei Martîniuc
asist. univ

Chișinău, 2026

Contents

1 Objective and Technical analysis

This laboratory work aims to design and implement a simple embedded system that allows user interaction with a hardware component (LED) via serial communication. The system will be structured using a layered architecture to ensure modularity and separation of concerns. The implementation will include drivers for controlling the LED and managing serial communication using `stdio.h` library by redirecting input/output streams, as well as a main program that processes user commands to control the LED state.

1.1 Serial communication

Serial communication is a method of transmitting data one bit at a time over a communication channel [1]. In this laboartory work, the communication will happen trough the serial interface of the Arduino board. The serial port is also called UART (Universal Asynchronous Receiver/Transmitter) and it allows for asynchronous communication between the microcontroller and a computer or other devices. Figure 1 illustrates the wiring of serial communicaton. As one can see, a serial bus consists of two main lines: the transmit line (TX) and the receive line (RX). The TX line is used to send data from the microcontroller to the computer, while the RX line is used to receive data from the computer to the microcontroller. In this laboratory work, we will use the serial communication to allow the user to send commands to control the LED state, and the microcontroller will respond accordingly by turning the LED on or off based on the received commands.

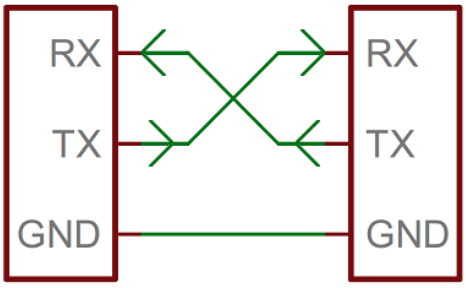


Figure 1: Serial communication bus

1.2 `stdio.h` library

The `stdio.h` library is a standard C library that provides functions for input and output operations, such as reading from the keyboard and writing to the console. In this laboratory work, we will utilize the `stdio.h` library to handle user input and output through the serial interface of the Arduino board. By redirecting the standard input and

output streams to the serial interface, we can use familiar functions like `printf()` and `scanf()` to interact with the user [2]. To redirect the standard streams, one can use the `fdev_setup_stream()` function to set up the UART streams for input and output, and then assign the standard input, output, and error streams. If `_RW` mode is enabled, the same stream can be used for both input and output.

1.3 Serial Terminal Emulator

A serial terminal emulator is a software application that allows users to interact with a serial device, such as an Arduino board, through a graphical interface. The core functionality of a emulator:

- **Serial Port Selection:** The emulator allows users to select the appropriate serial port to connect to the device. This is crucial for establishing communication between the computer and the microcontroller.
- **Baud Rate Configuration:** Users can configure the baud rate, which is the speed of data transmission. The baud rate must match the settings of the microcontroller for successful communication.
- **Data Transmission:** The emulator provides an interface for sending data to the microcontroller and receiving data from it. This allows users to interact with the device in real-time, sending commands and receiving responses.
- **Data Display:** The emulator displays the incoming data from the microcontroller in a readable format, allowing users to monitor the communication and debug their code effectively.

Examples of popular serial terminal emulators include PuTTY, Tera Term, RealTerm, CoolTerm, minicom, GNU screen, GtkTerm, and the built-in Serial Monitor in the Arduino IDE and the PlatformIO Serial Monitor [10]. These tools are essential for testing and debugging embedded systems that rely on serial communication.

1.4 Visual Studio Code extension `Platform.IO`

Platform.IO is an open-source ecosystem for IoT development that provides a unified interface for building, testing, and deploying embedded applications [8]. It integrates with Visual Studio Code to offer a powerful development environment for embedded systems. Key features of Platform.IO include:

- **Multi-platform Support:** Platform.IO supports a wide range of microcontroller platforms, including Arduino, ESP32, STM32, and many others. This allows devel-

opers to work with their preferred hardware without needing to switch development environments.

- **Library Management:** Platform.IO provides a built-in library manager that allows developers to easily search for, install, and manage libraries for their projects. This simplifies the process of adding functionality to embedded applications.
- **Build System:** Platform.IO uses a powerful build system that supports multiple build configurations and allows for easy integration with continuous integration tools. This helps streamline the development process and ensures that code is consistently built and tested.
- **Serial Monitor:** Platform.IO includes a built-in serial monitor that allows developers to interact with their embedded devices in real-time. This is particularly useful for debugging and testing applications that rely on serial communication.

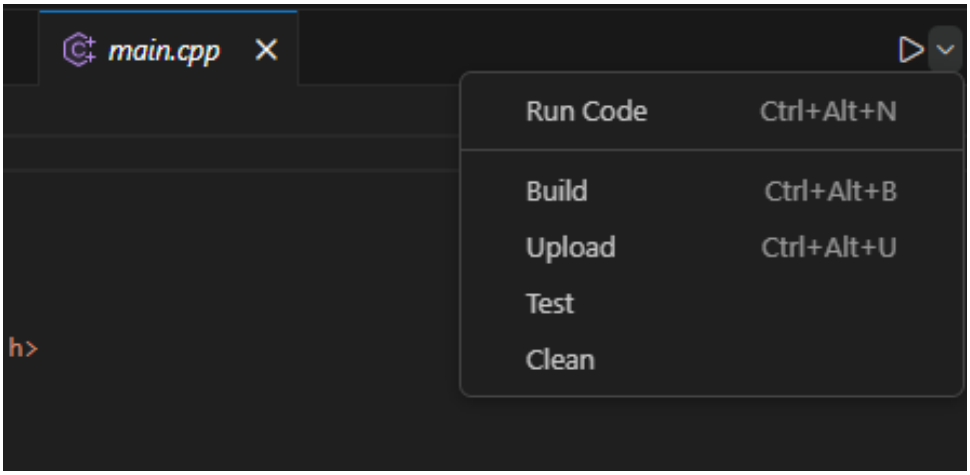


Figure 2: Platform.IO extension for Visual Studio Code

1.5 Hardware emulation with SimulIDE

The physical component of this laboratory work is an LED connected to an Arduino board. In real hardware implementation, the LED would be connected to a digital output pin of the Arduino through a current-limiting resistor. However, for testing and demonstration purposes, we can use SimulIDE, a simple real-time electronics simulator [9], represented in Figure 13. SimulIDE allows us to create virtual circuits and simulate their behavior without needing physical hardware. In SimulIDE, we can set up a virtual circuit that includes an Arduino board and an LED, and we can test our code by sending commands through the serial monitor to control the LED state. This provides a convenient way to verify the functionality of our code and the design of our system before deploying it on actual hardware. SimulIDE also allows uploading the real `.hex`

file generated by the Platform.IO build process, which means we can test the exact same code that will run on the physical Arduino board.

1.6 LED

An LED (Light Emitting Diode) is a semiconductor device that emits light when an electric current passes through it. In this laboratory work, the LED serves as the output device that we will control using the Arduino board. There is only command provided by Arduino API to control the LED - `digitalWrite()`, which allows us to set the state of a digital pin to either HIGH (turning the LED on) or LOW (turning the LED off). The LED is connected to a digital output pin of the Arduino through a current-limiting resistor to prevent damage to both the LED and the microcontroller. By sending commands through serial communication, we can control the state of the LED, allowing it to turn on or off based on user input.

Figure 3 shows an example of an LED connected to an electronic circuit, which is similar to the setup we will use in this laboratory work. The LED is typically connected in series with a resistor to limit the current flowing to 20mA [7] through it, ensuring that it operates within safe parameters. The digital output pin of the Arduino will be connected to the anode of the LED, while the cathode will be connected to ground through the resistor. This configuration allows us to control the LED state by setting the digital pin HIGH or LOW using the Arduino API.

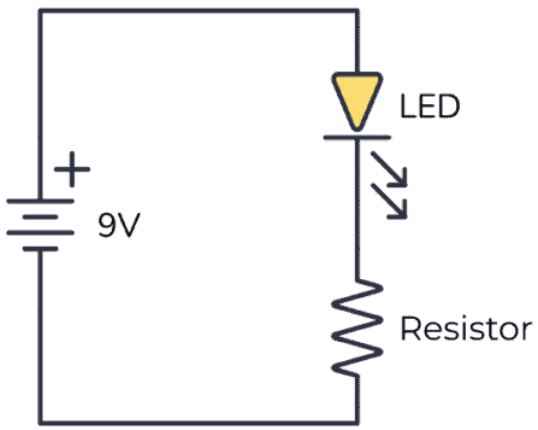


Figure 3: LED on an electronic circuit

Note: most common Arduino boards have a built-in LED connected to the D13 pin, which can be used for testing purposes without needing to set up an external circuit.

2 System design

2.1 System architecture

The system is designed to control an LED connected to an Arduino board via serial communication. The schema of the system architecture is presented in Figure 4. The main components of the system include:

- **Arduino Board:** The microcontroller that interfaces with the LED and handles serial communication.
- **LED:** The output device that will be turned on or off based on commands received via serial communication.
- **Serial Communication Interface:** Facilitates the exchange of commands between the user and the Arduino board.
- **User Input:** The user sends commands ("led on" or "led off") through a serial terminal to control the LED state.

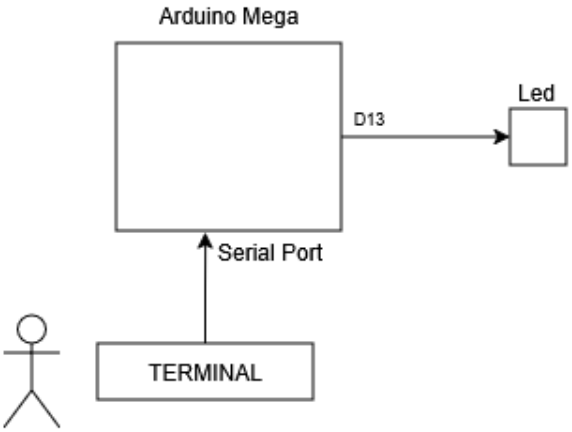


Figure 4: System Architecture Diagram

2.2 Flowchart of the program

The flowchart in Figure 5 illustrates the logic of the program running on the Arduino board. The program continuously listens for user input via serial communication and processes commands to control the LED state.

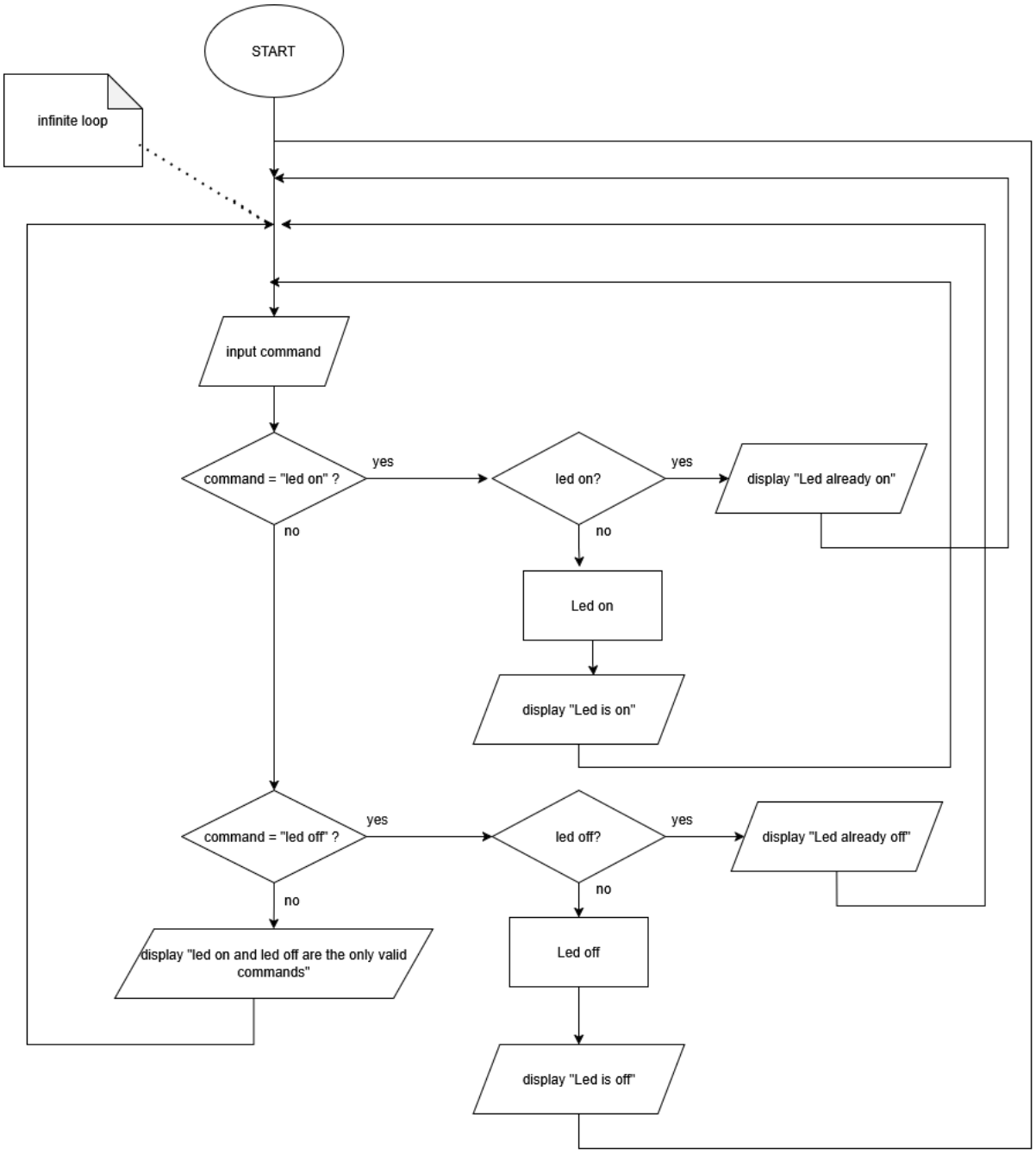


Figure 5: Program Flowchart

In this flowchart, there is an infinite loop that waits for user input. When a command is received, it checks if the command is "led on" or "led off" and turns the LED on or off accordingly. If the command is unrecognized, it prompts the user to enter a valid command.

2.3 Electronic circuit

Figure 6 illustrates the electronic circuit designed for controlling an LED using an Arduino board. The circuit consists of an LED connected to a digital output pin of the

Arduino through a current-limiting resistor. The ground pin of the Arduino is connected to the cathode of the LED, while the anode is connected to the resistor, which in turn is connected to the designated digital pin.

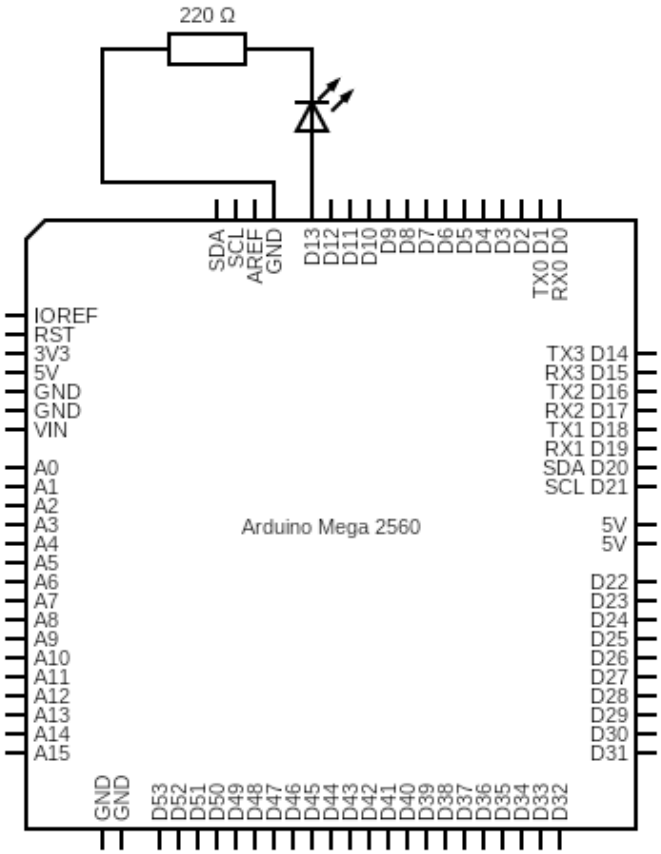


Figure 6: Circuit diagram of the LED control system

Since the Arduino board operates at 5V, a resistor value of 220Ω is chosen to limit the current through the LED to a safe level, preventing damage to both the LED and the Arduino pin.

Following the Ohm's law, the current flowing through the LED can be calculated as:

$$I = \frac{V_{supply} - V_{LED}}{R} = \frac{5V - 2V}{220\Omega} \approx 13.64mA$$

which is within the safe operating limits for both the LED and the Arduino pin.

Resistors with that value are marked with those color bands:

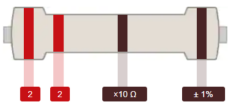


Figure 7: 220Ω resistor with color bands

2.4 Layered design (hardware/software)

2.4.1 LED Layered design

Figure 8 presents the layered design for the LED control module. The design is structured into six layers:

- **APP** - outermost application layer, uses higher-level functions to control the LED such as `led.on()`
- **SRV** - service layer, provides services to the application layer and interacts with the lower layers. For example, it may include functions to manage LED states.
- **ECAL** - embedded component abstraction layer, abstracts the hardware specifics of the LED, providing a uniform interface for higher layers.
- **MCAL** - microcontroller abstraction layer, interfaces directly with the microcontroller's hardware registers to control the LED.
- **MCU** - microcontroller layer, represents the actual microcontroller hardware that controls the LED. Here - D13 pin of Arduino.
- **ECU** - embedded component unit layer, represents the physical LED component connected to the microcontroller.

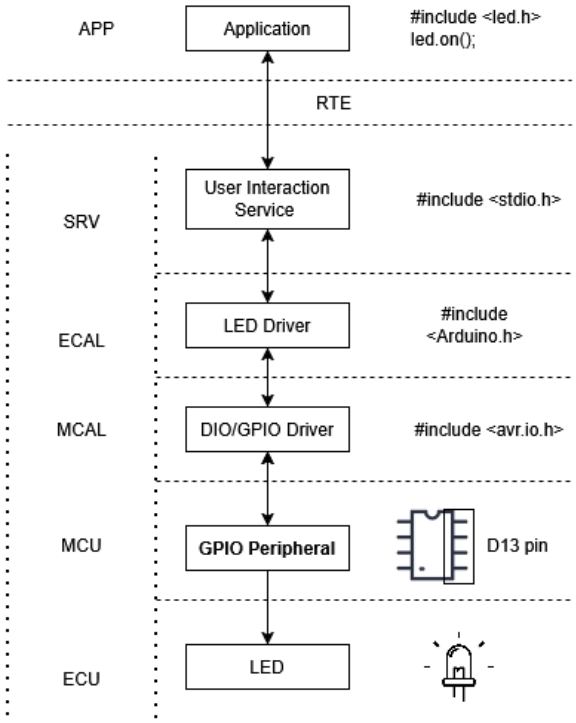


Figure 8: LED Layered Design

2.4.2 Serial I/O Layered design

Figure 9 illustrates the layered design for the Serial I/O module. The design is structured into six layers:

- **APP** - outermost application layer, uses higher-level functions to handle serial communication such as `readCommand()`
- **SRV** - service layer, provides services to the application layer and interacts with the lower layers. For example, it may include functions to manage serial data.
- **ECAL** - embedded component abstraction layer, abstracts the hardware specifics of the serial communication, providing a uniform interface for higher layers.
- **MCAL** - microcontroller abstraction layer, interfaces directly with the microcontroller's hardware registers to manage serial communication.
- **MCU** - microcontroller layer, represents the actual microcontroller hardware that handles serial communication. Here - UART module of Arduino.
- **ECU** - embedded component unit layer, represents the physical serial communication interface connected to the microcontroller, such as user terminal

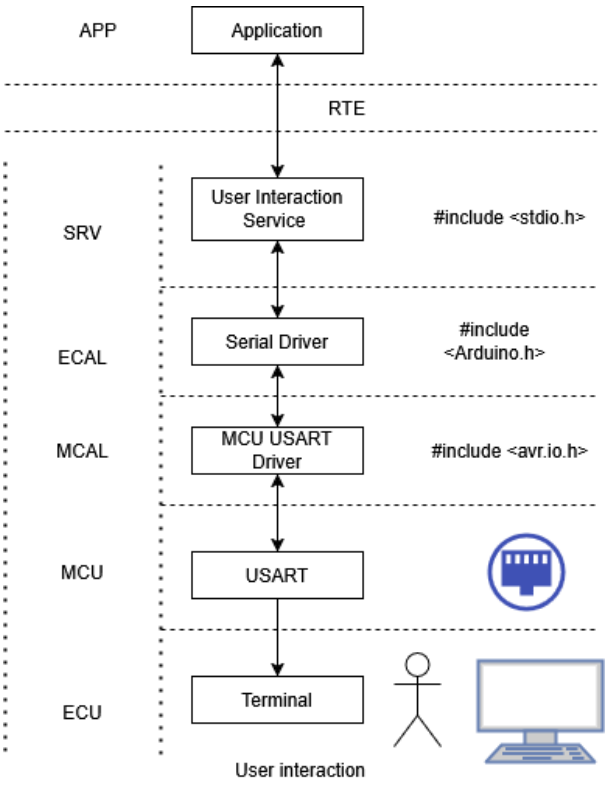


Figure 9: Serial I/O Layered Design

3 Implementation and results

3.1 Memory usage

Since the physical demonstration of the system is performed on a MCU with limited resources, it is crucial to analyze the memory usage of the implemented code. The memory usage can be categorized into two main types: Flash memory (program storage) and SRAM (data storage). Arduino Uno and Arduino Nano both have 32 KB of Flash memory and 2 KB of SRAM [4]. Hence, the implemented code should be optimized to fit within these constraints. Since the source code is relatively simple, most of the Flash memory will be empty and can be used for storing the string constants. Thankfully, `stdio` library provides a way to store string constants in Flash memory using the `PSTR()` macro and `printf_P()` function, which helps to save SRAM. The actual memory usage can be determined by compiling the code and checking the output provided by the Platform.IO, which shows the percentage of Flash and SRAM used.

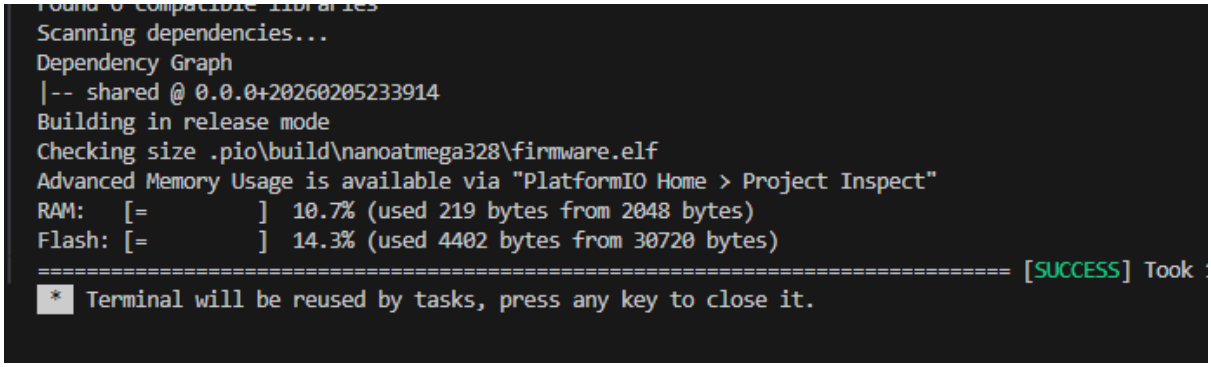


Figure 10: Memory usage of the implemented code

3.2 Driver implementation

The driver implementation for the LED control system consists of two main components: the LED driver and the Serial I/O driver. The LED driver is responsible for controlling the state of the LED, providing functions to turn it on and off. The Serial I/O driver manages the communication between the user and the Arduino board, allowing the user to send commands to control the LED. The implementation is structured in a layered design, as described in the previous section, to ensure modularity and separation of concerns. The source code for both drivers is provided in the appendix, with the LED driver containing functions such as `led.on()` and `led.off()`, and the Serial I/O driver containing function to redirect Serial IO stream to STDIO. The source code of both drivers is available in the GitHub repository [3] and in Appendix for both led 7 and serialio 5.

Led driver implementation, simplified, is represented on figure 11. As mentioned, it

has a high level interface for the application layer, which abstracts the hardware specifics of controlling the LED provided by `digitalWrite()` function of the Arduino API.

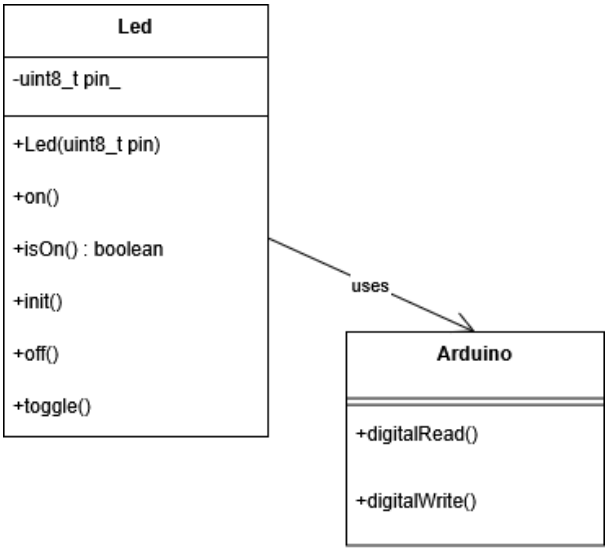


Figure 11: LED driver class diagram

Simplified Serial I/O driver implementation is shown on figure 12. It provides a function to redirect the standard input and output streams to the serial interface of the Arduino board, allowing the use of standard I/O functions for communication.

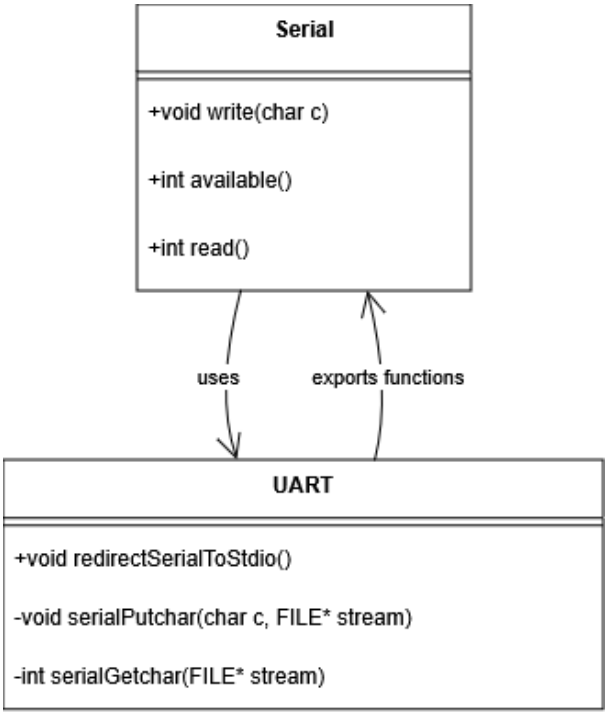
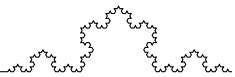


Figure 12: Serial I/O driver class diagram



3.3 Main program implementation

The entry point of the program is the `main.cpp` file 1, which contains the main loop that listens for user input via serial communication and processes commands to control the LED state. The program initializes the Serial communication and the LED driver, then enters an infinite loop where it waits for user input. When a command is received, it checks if the command is "led on" or "led off" and calls the appropriate functions from the LED driver to change the state of the LED. If an unrecognized command is received, it prompts the user to enter a valid command. The main program is designed to be simple and efficient, ensuring that it can run on microcontrollers with limited resources while providing a responsive user experience.

To keep the main function minimal, a `LedController` file is implemented, which contains the logic for processing user commands and controlling the LED state. This separation of concerns allows for better code organization and maintainability. The main function simply initializes the necessary components and calls the `LedController` to handle the user input and control the LED accordingly.

3.4 Simulation

The implemented system successfully allows the user to control the LED state via serial commands. When the user inputs "led on", the LED lights up, and when "led off" is entered, the LED turns off. The system responds correctly to valid commands and prompts the user for valid input when unrecognized commands are received.

The steps shows in the figures below are the following:

1. Figure 13 shows the SimulIDE project setup for the LED control system.
2. Figure 14 displays the serial monitor prompting the user for input.
3. Figure 15 illustrates the LED turned on after receiving the "led on" command.
4. Figure 16 shows the serial monitor indicating that the LED is already on when the "led on" command is sent again.
5. Figure 17 depicts the LED turned off after receiving the "led off" command.
6. Figure 18 presents the serial monitor displaying an invalid command message when an unrecognized command is entered.

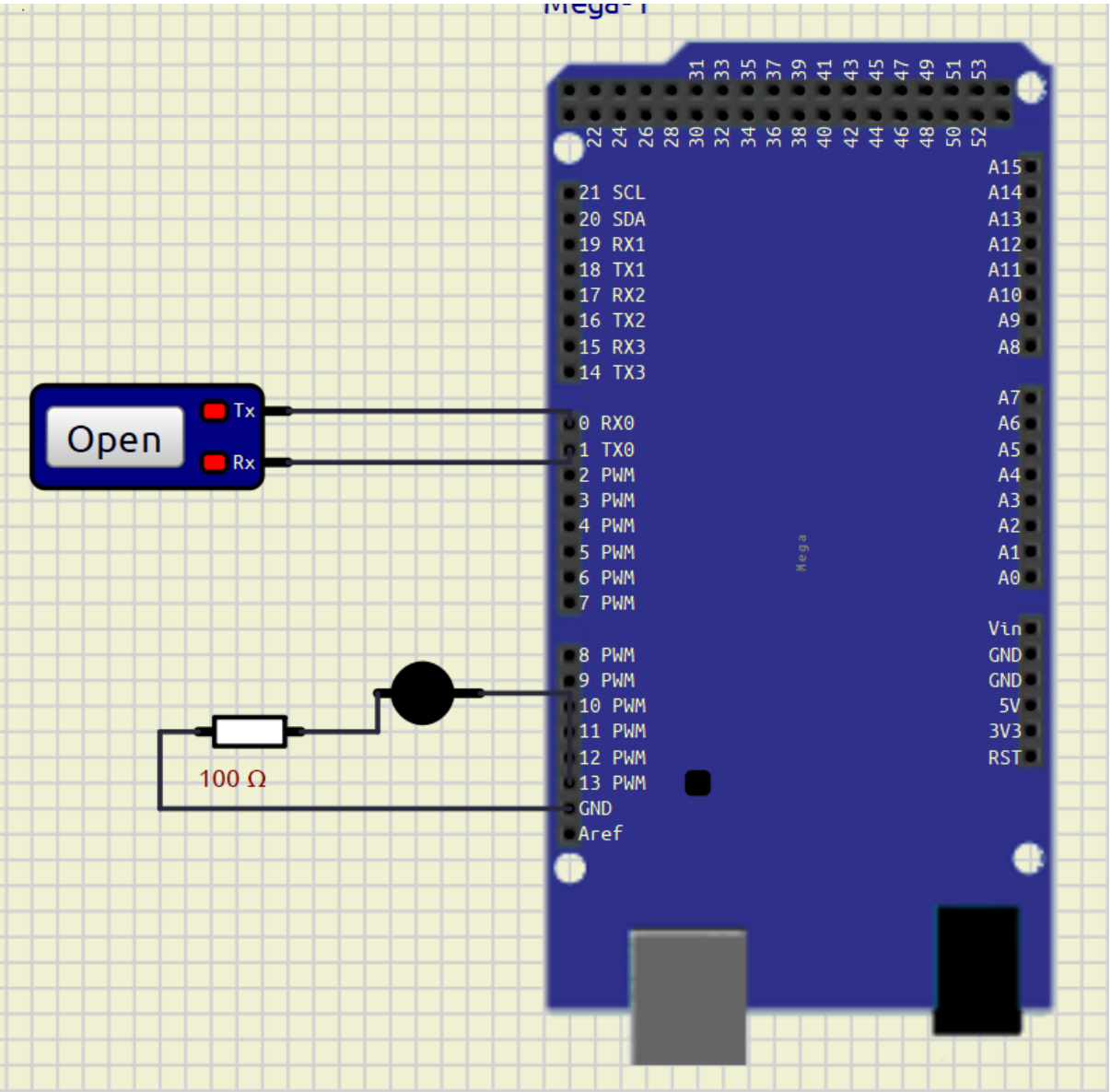


Figure 13: SimulIDE project setup for LED control

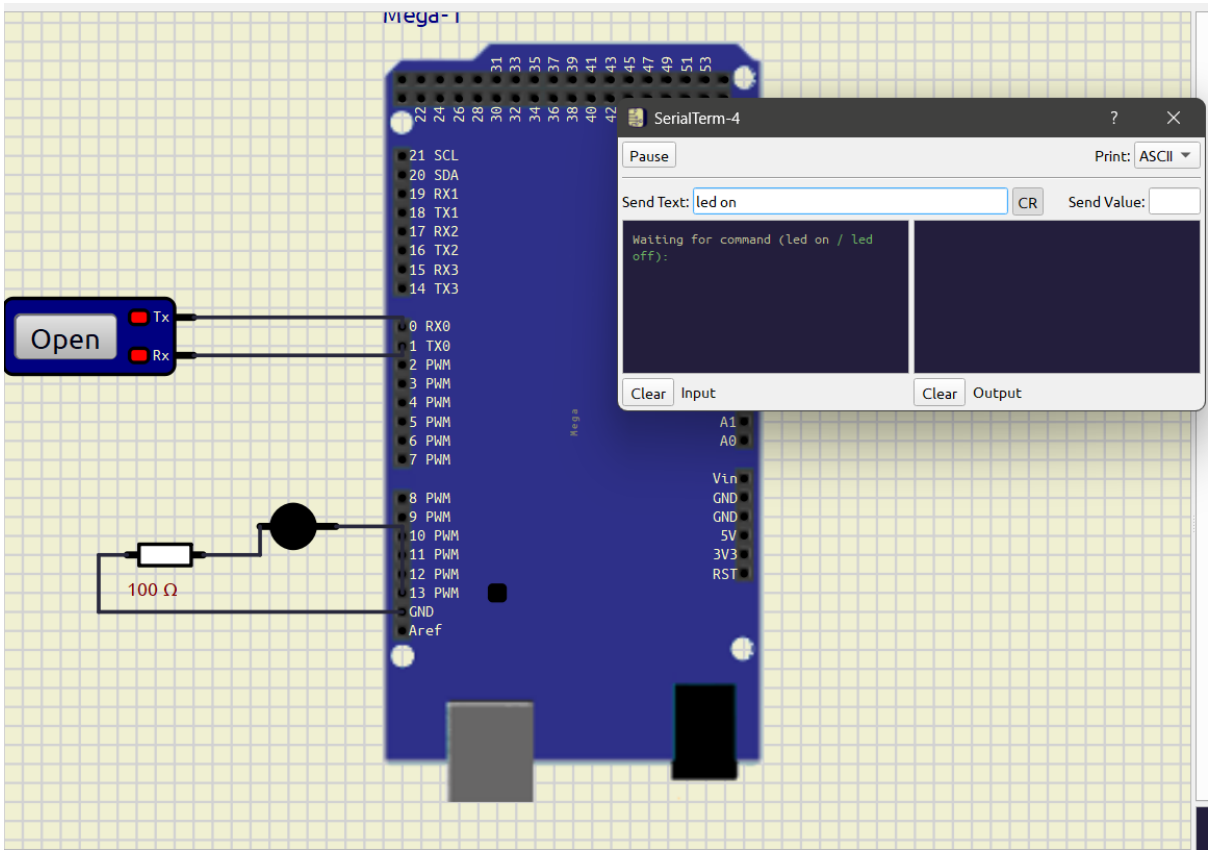


Figure 14: Serial monitor prompting for input

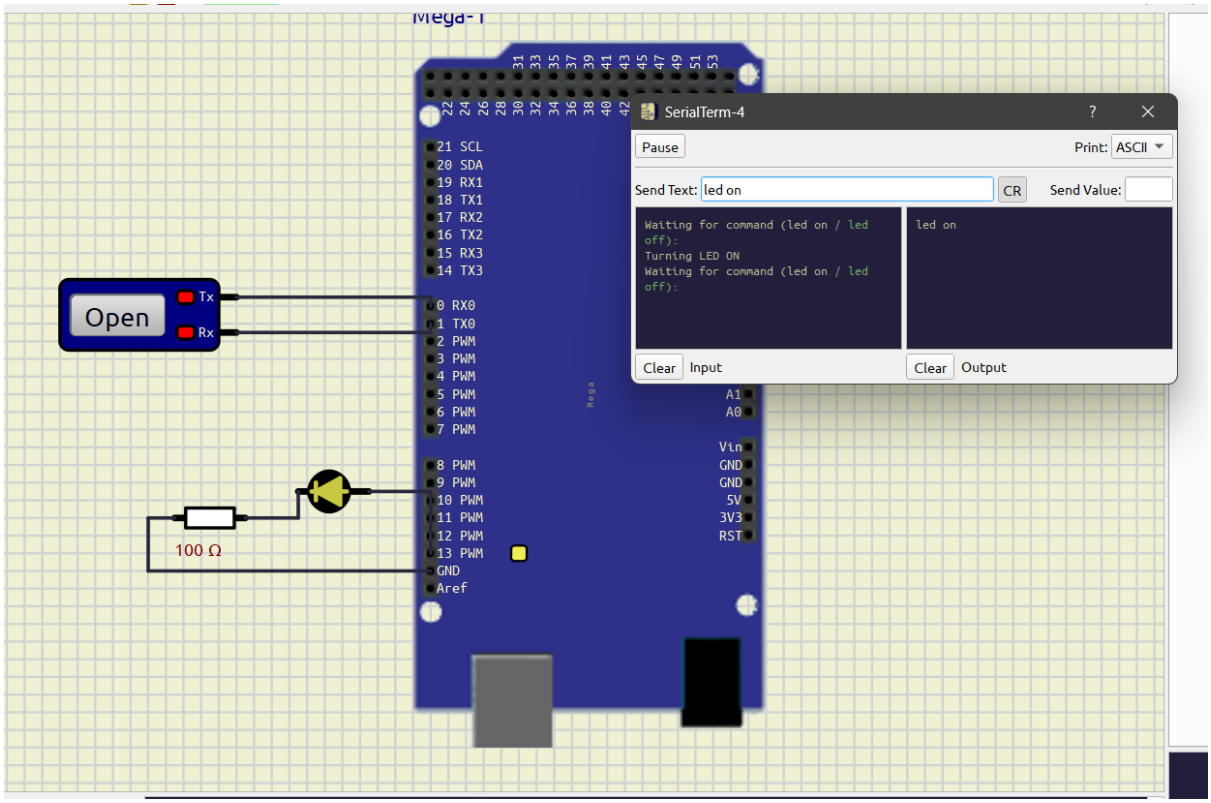


Figure 15: LED turned on after receiving "led on" command

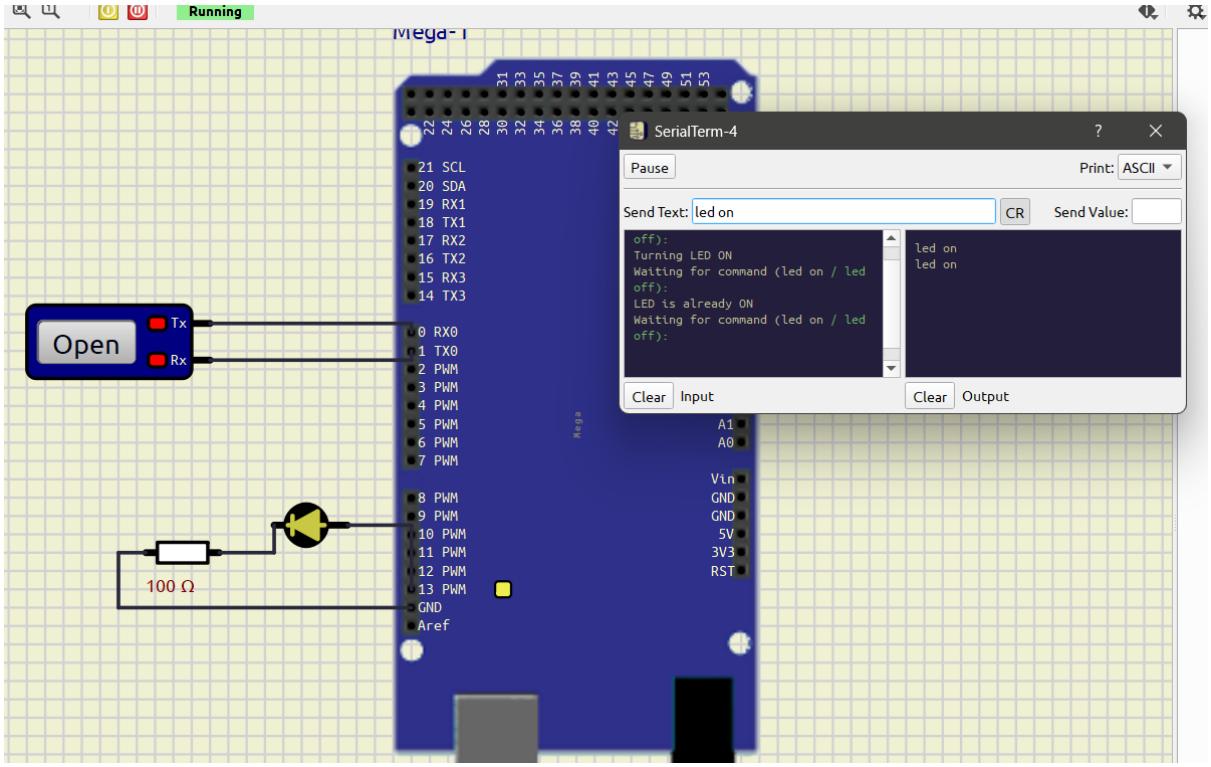


Figure 16: Serial monitor showing that LED is already on

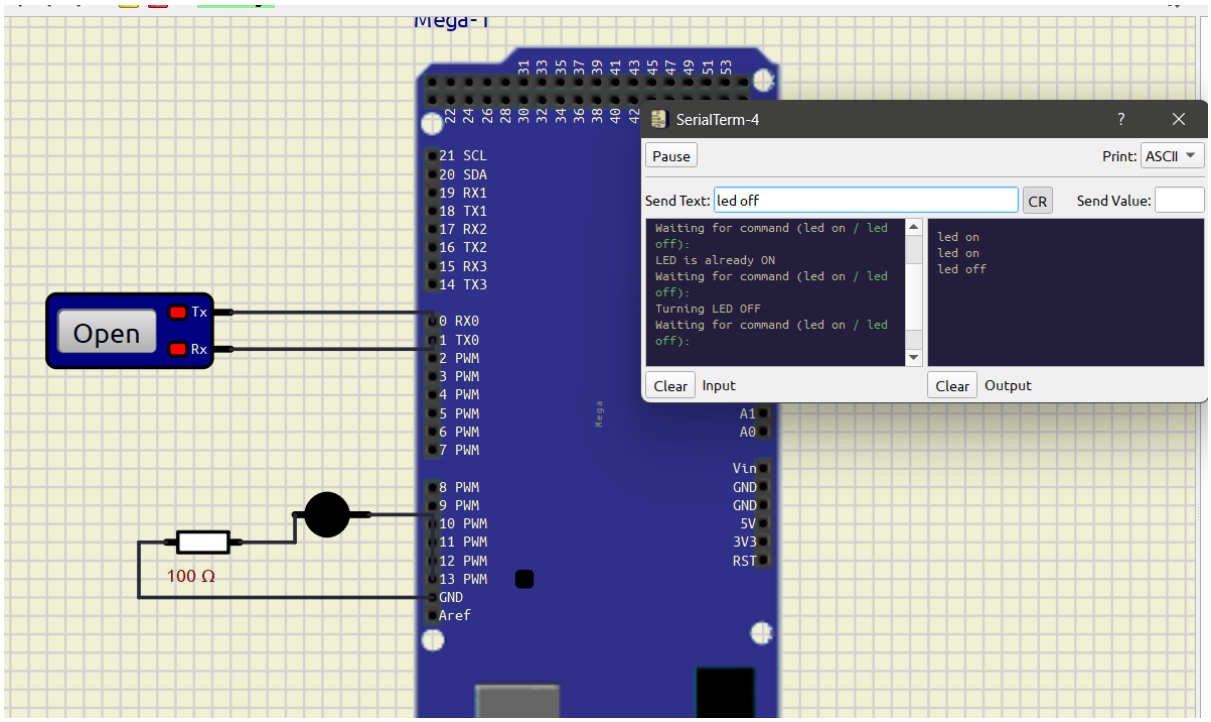


Figure 17: LED turned off after receiving "led off" command

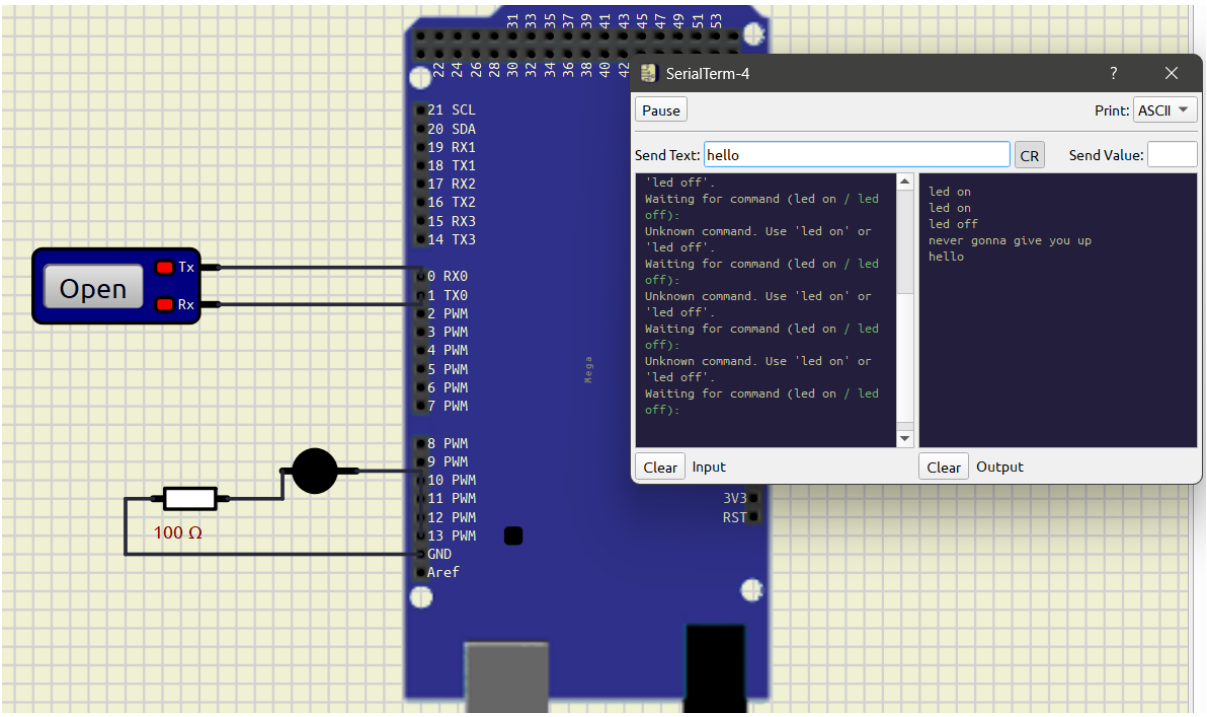


Figure 18: Serial monitor showing invalid command message

4 Conclusion

During this laboratory work I implemented a simply embedded system which allows user interaction with hardware component (LED) via serial communication. The system is designed with a layered architecture to ensure modularity and separation of concerns, making it easier to maintain and extend in the future. The implementation includes a driver for controlling the LED and a driver for managing serial communication, both of which are structured in a way that abstracts hardware specifics from the application layer. These drives are stored in separate files indepdendent of the project and can be included in the `Platform.IO` project with a simple directive of `.ino` file: `lib_deps = ../../shared`. The main program initializes the necessary components and continuously listens for user input to control the LED state. The system successfully responds to valid commands and provides feedback for invalid commands, demonstrating effective user interaction with the hardware component.

For this communication, I redirect `stdin` and `stdout` to the serial interface of the Arduino board, allowing seamless input and output operations through the serial monitor. Despite the memory usage consequences of using standard I/O functions, the implementation remains efficient and fits within the constraints of the microcontroller one of the reason being the usage of `F()` macro for string constants to save SRAM.

Overall, this laboratory work provides a solid foundation for understanding embedded systems design and implementation, particularly in the context of user interaction with hardware components via serial communication.

References

[1] Serial Communication https://learn.sparkfun.com/tutorials/serial-communication/all

[2] AVR-Libc Reference Manual - Standard I/O https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html

[3] GitHub repository https://github.com/TimurCravtov/EmbeddedSystemsLabs

[4] Arduino Memory specification and architecture https://docs.arduino.cc/learn/programming/memory-guide/

[5] Arduino LED Control https://roboticsbackend.com/arduino-led-complete-tutorial/

[6] Circuit Diagram Builder https://www.circuit-diagram.org/docs

[7] Standard 5mm LED specification https://www.make-it.ca/5mm-led-specifications/

[8] Platform.IO - Embedded Development Ecosystem https://platformio.org/

[9] SimulIDE - Simple real time electronics simulator https://simulide.com/p/mcus-/l

[10] Serial terminal emulator overview - Toradex Developer Center https://developer.toradex.com/software/development-resources/serial-terminal-emulator/

A Source Code

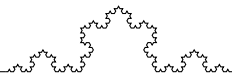
Besides this appendix, the source code is available in the GitHub repository [3] with all build instructions.

main.cpp

```cpp
#include <Arduino.h>

#include <stdio.h>
#include <stdlib.h>
#include <led/led.h>
#include <serialio/serialio.h>
#include <string.h>
#include <LedController.h>

```



```
10 // set led pin number and led object
11 constexpr uint8_t ledPinNum = 3;
12 Led led(ledPinNum);
13
14 void setup() {
15
16   led.init();
17   // start serial communication
18   Serial.begin(9600);
19   delay(1000);
20
21   redirectSerialToStdio();
22 }
23
24 void loop() {
25
26     // reading command from serial
27     char buffer[10] = {0};
28
29     // prompt to read data
30     printf_P(PSTR("Waiting for command (led on / led off): \n> "
           ));
31
32     // read the line
33     // TODO: handle backspace operator
34     scanf(" %9[^\n\r]", buffer);
35     int c;
36
37     // clear the buffer
38     while ((c = getchar()) != '\n' && c != '\r' && c != EOF);
39
40     // process command introduced by user
41     processCommand(led, buffer);
42
43 }
```

Listing 1: main.cpp

`LedController.h`

```
1 #pragma once
```

```
2
3 #include <led/led.h>
4
5 void processCommand(Led& led, char* command);
```

Listing 2: LedController.h

LedController.cpp

```
1  #include <LedController.h>
2  #include <string.h>
3
4  void handleLed(Led& led, bool);
5
6  void processCommand(Led& led, char* command) {
7
8      // printf_P(PSTR("Received command: %s\n"), command);
9
10     if (strcasecmp_P(command, PSTR("led on")) == 0) {
11         handleLed(led, true);
12     } else if (strcasecmp_P(command, PSTR("led off")) == 0) {
13         handleLed(led, false);
14     } else {
15         printf_P(PSTR("Unknown command. Use 'led on' or 'led off
               '.\n"));
16     }
17 }
18
19 void handleLed(Led& led, bool turnOn) {
20     if (turnOn) {
21         if (!led.isOn()) {
22             led.on();
23             printf_P(PSTR("LED turned ON\n"));
24         } else {
25             printf_P(PSTR("LED is already ON\n"));
26         }
27     } else {
28         if (led.isOn()) {
29             led.off();
30             printf_P(PSTR("LED turned OFF\n"));
31         } else {
```

```
32                printf_P(PSTR("LED is already OFF\n"));
33            }
34        }
35    }
```

Listing 3: LedController.cpp

serialio.h

```
1  #pragma once
2
3  #include <Arduino.h>
4
5  // this function does exactly what do you think it does
6  void redirectSerialToStdio();
```

Listing 4: serialio.h

serialio.cpp

```
1  #include <Arduino.h>
2  #include <serialio/serialio.h>
3  #include <stdio.h>
4
5  #include <stdlib.h>
6
7
8  // add a char to serial output
9  int serialPutchar(char c, FILE* stream) {
10   Serial.write(c);
11   return 0;
12 }
13
14 // get a char from serial input
15 int serialGetchar(FILE* stream) {
16   while (!Serial.available());
17   return Serial.read();
18 }
19
20 // Helper functions for redirecting Serial to stdio
21 void redirectSerialToStdio() {
22   static FILE uartinout;
```

```
23
24    fdev_setup_stream(&uartinout, serialPutchar, serialGetchar,
          _FDEV_SETUP_RW);
25
26    stdout = stdin = stderr = &uartinout;
27  }
```

Listing 5: serialio.cpp

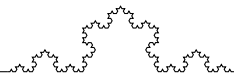
`led.h`

```
1  #pragma once
2
3  #include <Arduino.h>
4
5  /// @brief A simple LED control class
6  class Led {
7  public:
8    explicit Led(uint8_t pin);
9
10   void on();
11   boolean isOn();
12   void init();
13   void off();
14   void toggle();
15
16 private:
17   uint8_t pin_;
18 };
```

Listing 6: led.h

`led.cpp`

```
1  #include <led/led.h>
2
3  Led::Led(uint8_t pin) : pin_(pin) {}
4
5  void Led::init() {
6    pinMode(pin_, OUTPUT);
7  }
8
```



```
9  void Led::on() {
10   digitalWrite(pin_, HIGH);
11 }
12
13 boolean Led::isOn() {
14   return digitalRead(pin_) == HIGH;
15 }
16
17 void Led::off() {
18   digitalWrite(pin_, LOW);
19 }
20
21 void Led::toggle() {
22   digitalWrite(pin_, !digitalRead(pin_));
23 }
```

Listing 7: led.cpp