Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 2.1

Discipline: Embedded Systems

Student: Revenco Victor, FAF - 221

Checked: asist. univ., Martiniuc A.

Chișinău 2025

Analysis of the Situation in the Field

1. Description of the Technologies Used and Application Context

The project employs an embedded system approach using a microcontroller unit (MCU) such as an Arduino Uno or ESP32. The software is developed using PlatformIO in Visual Studio Code, implementing non-preemptive task scheduling. The communication between tasks follows the provider-consumer model, ensuring data integrity and synchronization.

Hardware Components

- **Microcontroller**: Arduino Uno / ESP32
- **LEDs**: Used to indicate different task states
- **Resistors (220Ω)**: Current limiting for LEDs
- **Push Buttons**: Used for user input to toggle LEDs and modify state variables
- **Breadboard**: For circuit prototyping
- **Jumper Wires**: For electrical connections
- **USB Power Supply**: For powering the MCU

Software Components

- **PlatformIO with Visual Studio Code**: Integrated Development Environment (IDE)
- **C++ for Embedded Systems**: Programming language used for implementation
- **STDIO Functions (printf)**: For system reporting and monitoring
- **Serial Monitor**: For displaying system status

2. System Architecture Explanation and Solution Justification

The system follows a sequential execution model with a non-preemptive task scheduler. The tasks are synchronized using global variables and signals, avoiding the need for a real-time operating system (RTOS). The scheduling methodology minimizes CPU load by setting appropriate recurrence rates and offsets for task execution.

The provider-consumer model is implemented as follows:

- **Task 1 (Button LED)**: Acts as a provider, modifying a shared state variable based on button input.
- **Task 2 (LED Intermittent)**: Acts as a consumer, toggling based on the state of Task 1.
- **Task 3 (State Variable Control)**: Acts as both provider and consumer, modifying a recurrence variable affecting Task 2.

- **Idle Task (System Reporting)**: Consumes global variables and prints system status to a monitor/LCD.

3. Case Study: LED Control in an Industrial Automation System

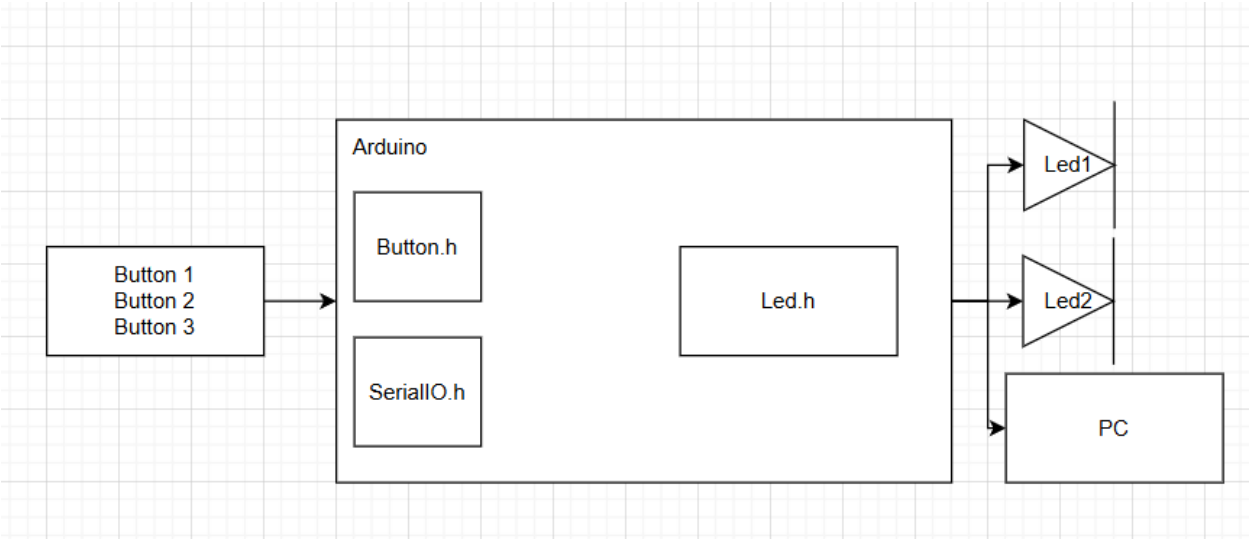
A practical demonstration of task scheduling is performed with the following constraints:

- The LED toggling task executes only when the first LED is off.
- The blinking rate is dynamically adjusted based on user input.
- A reporting mechanism continuously displays the current state.

Design

1. Architectural Sketch and Component Interconnection

- Buttons connected to digital input pins
- LEDs connected to output pins with resistors
- Microcontroller executing sequential task scheduling
- Serial Monitor or LCD for system status output

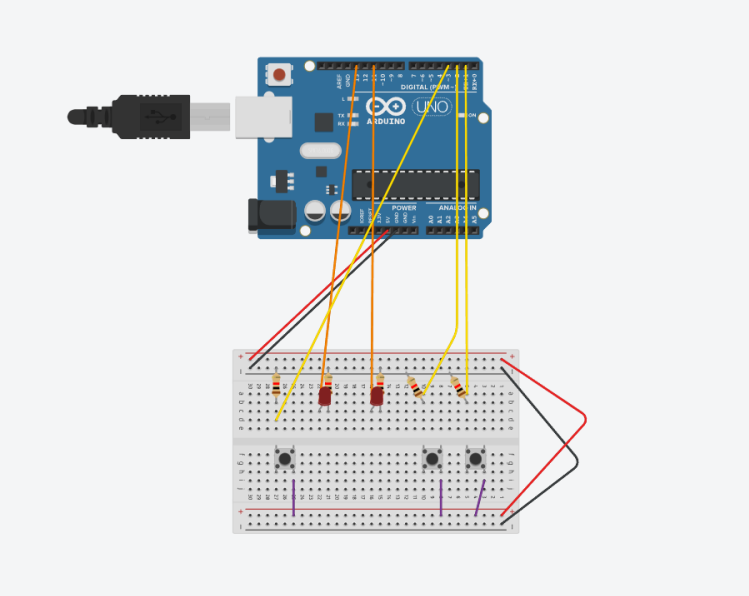


Component Description and Their Roles:

- **Microcontroller**: Core processing unit executing tasks

- **LEDs**: Provide visual feedback for task states
- **Buttons**: User input for system state changes
- **Resistors**: Current limiting for LED protection

The sketch shows how the components interact to enter a password and turn on/off the LED with serial commands.



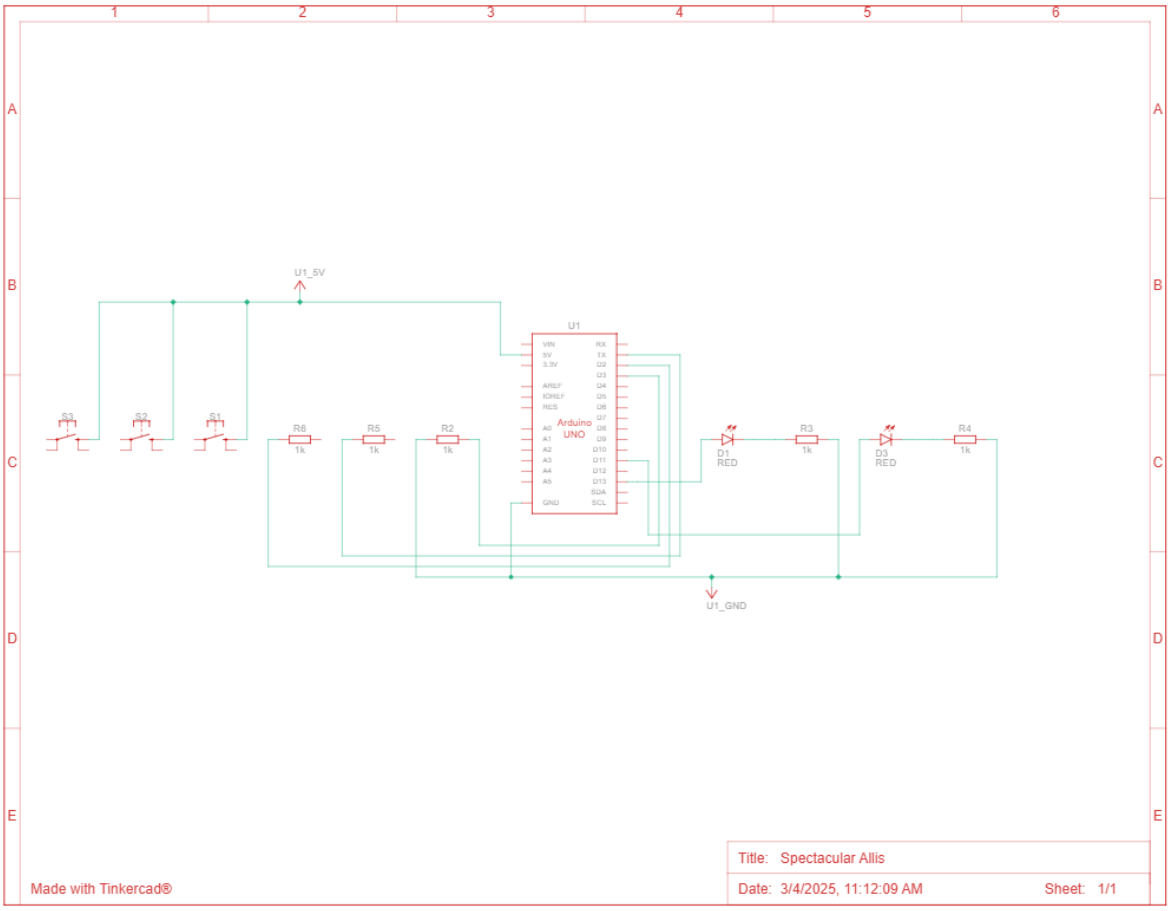


Figure 1, 2 Electrical schematic

This image shows an **Arduino Uno** connected to a breadboard circuit, which includes LEDs, resistors, and push buttons. Here's a breakdown of the components and their connections:

1. Microcontroller (Arduino Uno)

- The **Arduino Uno** is the main processing unit, providing power and controlling the circuit.

2. LEDs (Light Emitting Diodes)

- There are **three LEDs** placed on the breadboard.
- Each LED is connected to a **current-limiting resistor** to prevent excessive current.
- The **other end of the resistors is connected to digital output pins** on the Arduino.

3. Push Buttons

- The breadboard includes **three push buttons**. These are used for **user input**, such as toggling LED states or adjusting variables.
- The buttons are connected in a **pull-down configuration**, meaning they pull the input **LOW (0V) when not pressed** and go **HIGH (5V) when pressed**.
- Wires connect the buttons to digital input pins on the Arduino.

4. Power Connections

- The Arduino **5V pin** is connected to the **red power rail** of the breadboard.
- The Arduino **GND (ground) pin** is connected to the **black power rail** of the breadboard.
- The components use these rails to share power.

2. Schematic diagrams

To understand the system's behavior, a Flowchart and a Finite State Machine (FSM) are used.

Flowchart – Serial Command Processing

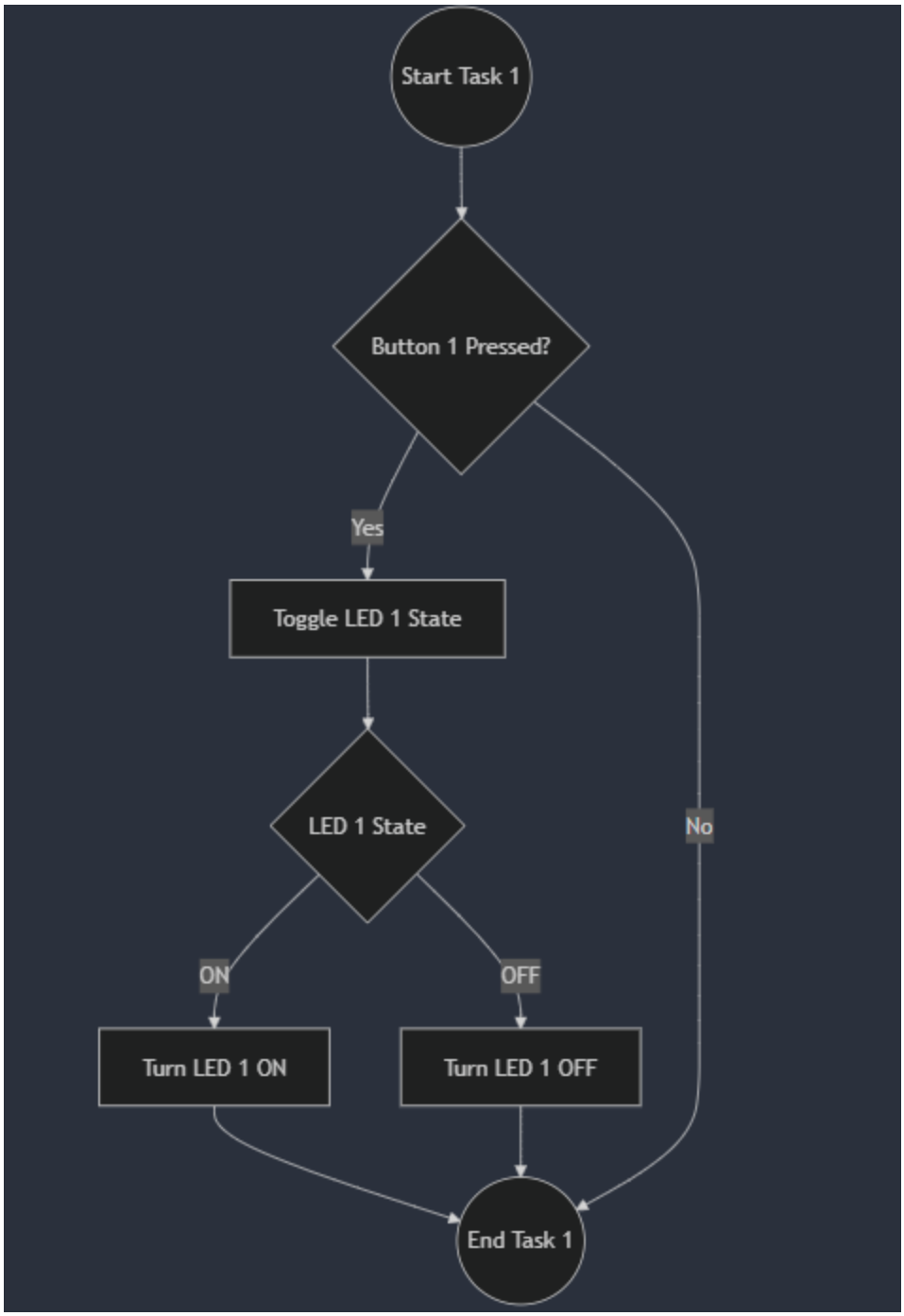


Figure 3 Flowchart task1

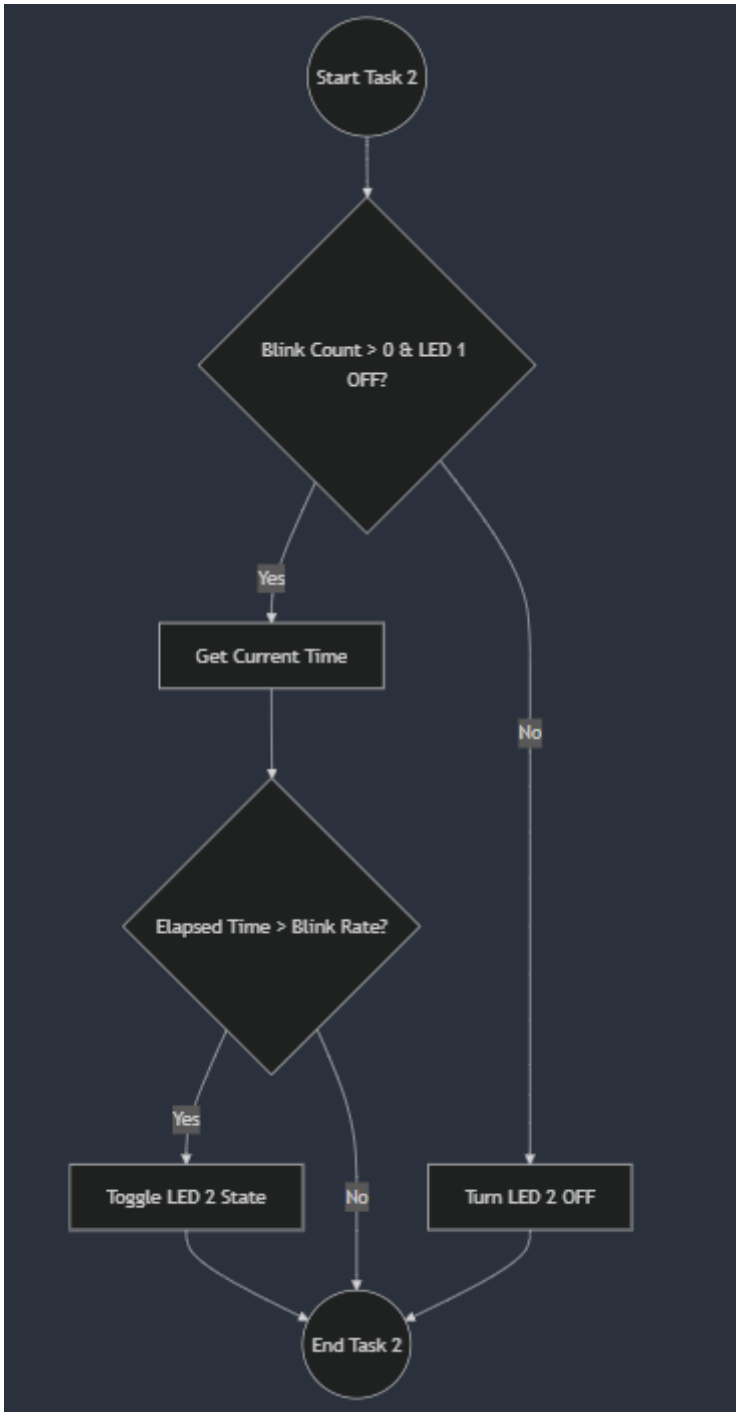


Figure 4 Flowchart task2

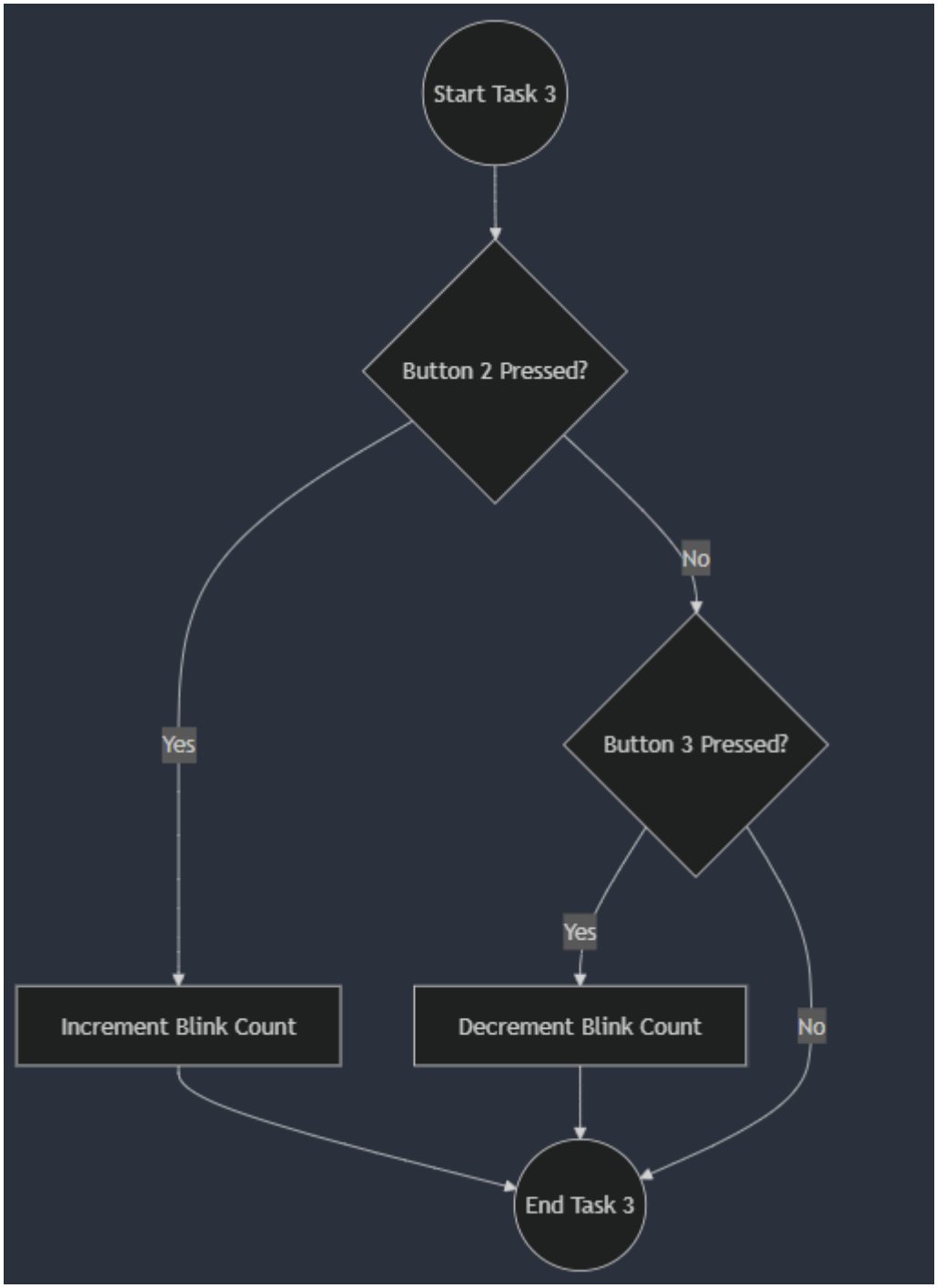


Figure 5 Flowchart task3

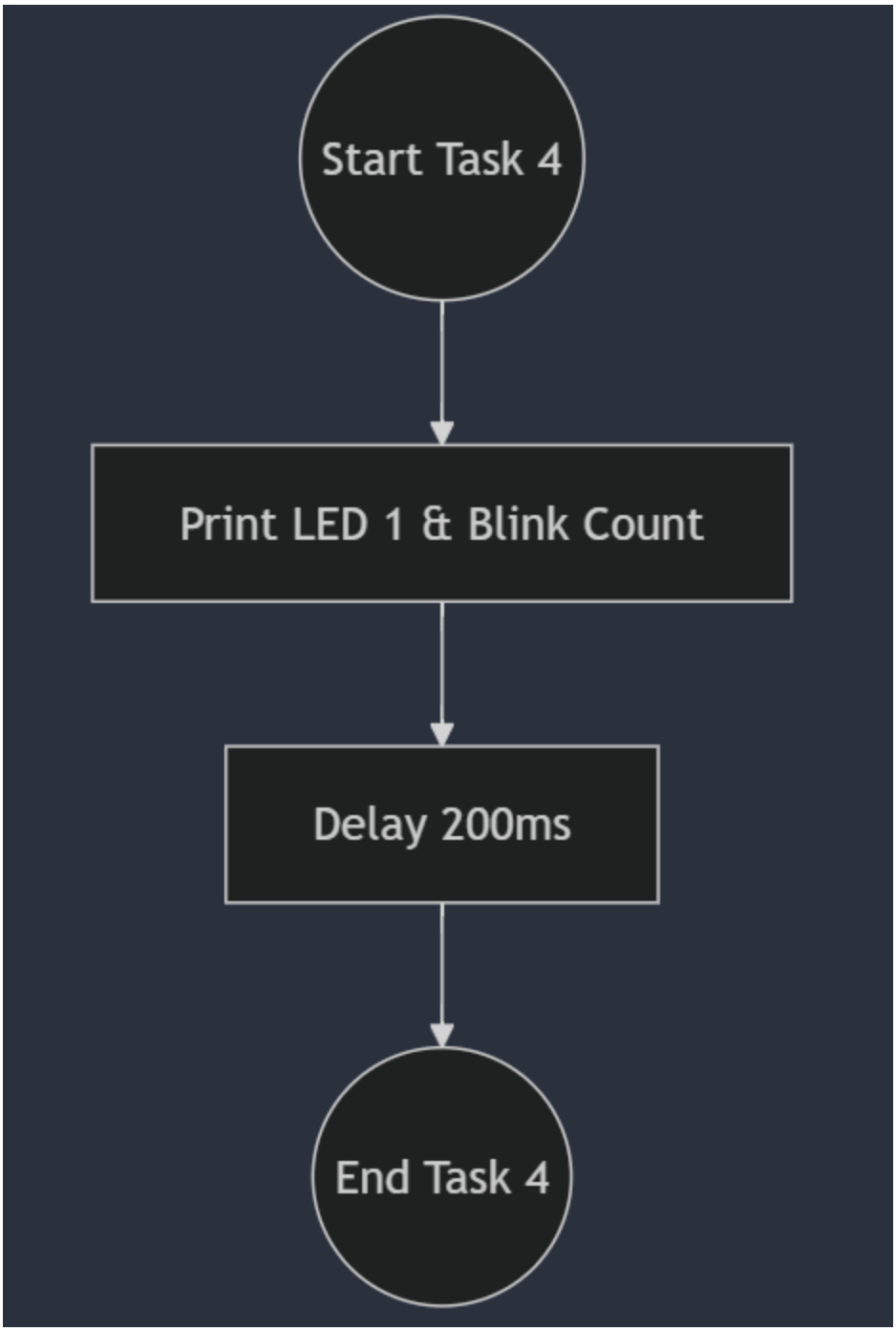


Figure 6 Idle

The FSM includes the following states:

1. Idle – Waits for commands from the user.
2. Processing – Processes the received command.

3. Led on – Turns on the LED.
4. Led off – Turns off the LED.
5. Counter++ – Increments counter.
6. Counter-- - Decrements counter.

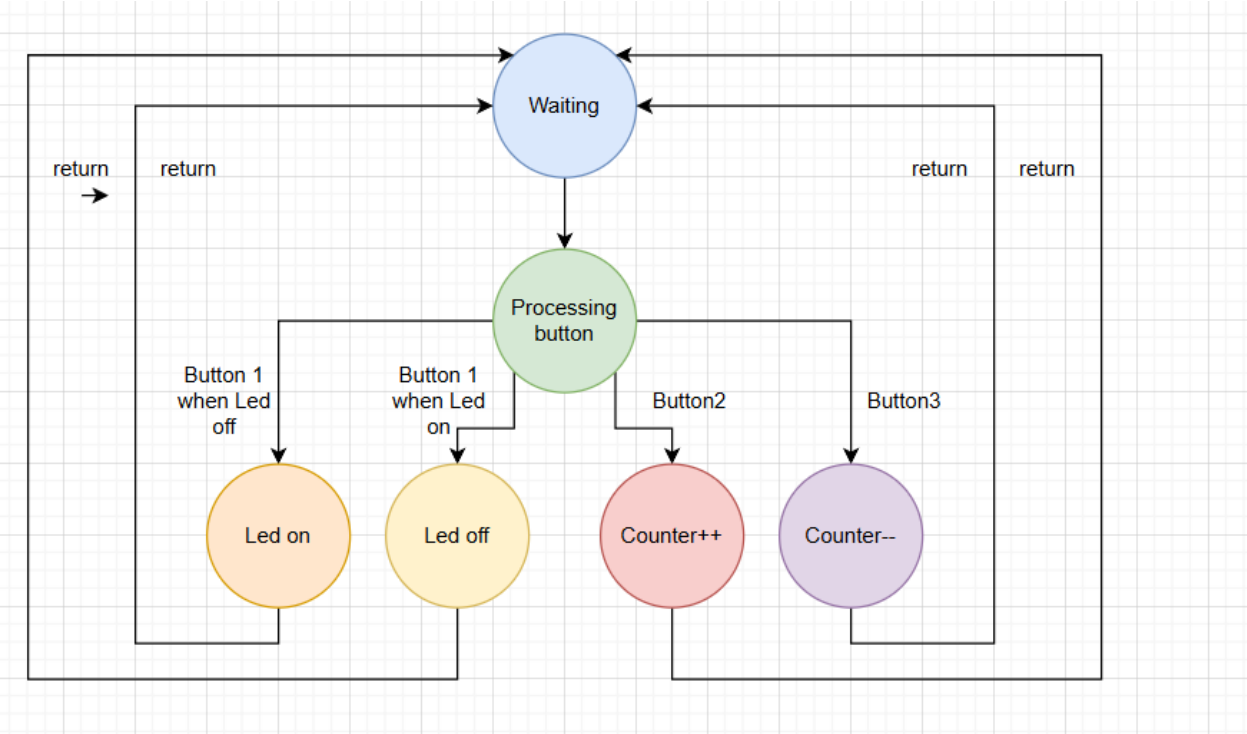


Figure 7 FSM diagram

3. Modular implementation

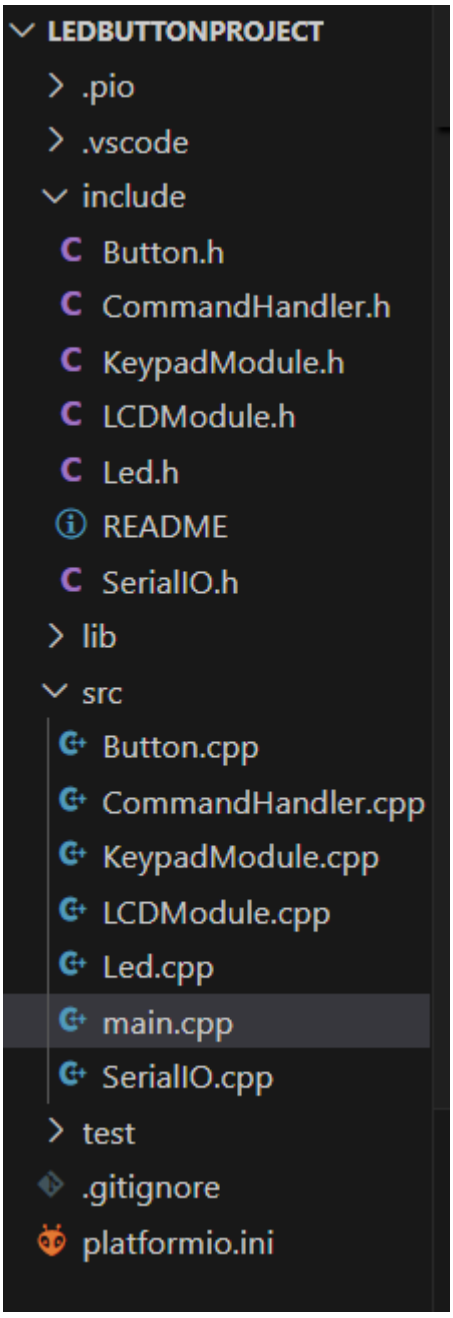


Figure5 Project organization

This header file, named *Led.h*, defines the interface for controlling an LED. It declares 3 functions without providing their implementations:

1. toggle() – This function is used to change the state of the LED (State: ON or OFF)
2. turnOn() – This function is used to change the state of the LED to ON
3. turnOff() – This function is used to change the state of the LED to OFF

In the SerialIO.h we have the following methods for reading and writing the text:

1. serial_putc(char c, FILE *stream) – it uses Serial.write to write the text
2. serial_getc(FILE *stream) – it uses Serial.read serial is not available

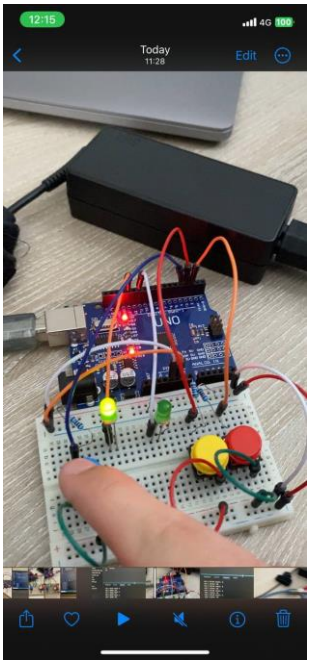
In the file Led.cpp we implement all methods in the Led.h:

1. toggle() – it changes the state using '!', and writes to the pin the state HIGH if its LOW and vice versa
2. turnOn() – sets the state to true and sets the pin to HIGH
3. turnOff() – sets the state to false and sets the pin to LOW

In the Button.h we have the following methods for reading button pressed:

1. Constructor – read button input

Results



https://drive.google.com/file/d/1mq9aQYGayB3Bn766FlOnAlqN_RGh9hlB/view?usp=sharing

Conclusions

This project demonstrates a structured approach to microcontroller-based task execution by implementing modular programming and a sequential task scheduler. The system effectively integrates multiple tasks using a provider-consumer model, ensuring efficiency and synchronization without the need for an RTOS. The use of non-blocking techniques, such as `millis()`, enhances responsiveness, allowing concurrent task execution without unnecessary delays.

By designing independent tasks, the architecture is flexible and scalable, enabling future expansion with additional sensors, actuators, or communication protocols. The user interface, consisting of push buttons and LEDs, provides an intuitive way to interact with the system, while the serial monitor ensures transparency in system behavior and debugging.

Furthermore, the electrical and logical schematics ensure a well-structured hardware-software integration, making this project a valuable example of embedded system design principles. The combination of structured task execution, modular design, and dynamic user control showcases best practices in microcontroller programming, making it a solid foundation for further development in IoT and automation applications.

Bibliography

1. Official Arduino Documentation
    - Arduino Reference – Serial Communication https://www.arduino.cc/reference/en/#communication
    - Arduino Mega 1280 Pinout & Datasheet https://docs.arduino.cc/hardware/mega-1280
2. PlatformIO Official Documentation
    - PlatformIO for Arduino Development https://docs.platformio.org/en/latest/platforms/atmelavr.html
3. TUM Courses
    - Introducere în Sistemele Embedded și Programarea Microcontrolerelor
    - Principiile comunicației seriale și utilizarea interfeței UART
4. https://drive.google.com/file/d/1mq9aQYGayB3Bn766FlOnAlqN_RGh9hlB/view?usp=sharing

Appendix

1. **GitHub**: https://github.com/Kipitokisk/SI_Lab

```
#include <Arduino.h>
#include "Button.h"
#include "LED.h"
#include "SerialIO.h"

// Button objects for task 1 and task 3
Button button1(4);  // Button for Task 1 (LED toggle)
Button button2(3);  // Button for Task 3 (Increment)
Button button3(2);  // Button for Task 3 (Decrement)

// LED objects for task 1 and task 2
Led led1(13);  // LED for Task 1 (Button LED)
Led led2(11);  // LED for Task 2 (Intermittent LED)

// Global variables
volatile bool led1State = false;  // State of LED 1 (Task 1)
volatile int blinkCount = 0;      // Number of blinks for Task 2
volatile bool led2State = false;  // State of LED 2 (Task 2)

// Task 1: Button LED
void taskButtonLED() {
    if (button1.isPressed()) {
        led1State = !led1State;  // Toggle LED 1 state
        if (led1State) {
            led1.turnOn();
        } else {
            led1.turnOff();
        }
    }
}

void taskLEDIntermittent() {
    static unsigned long previousMillis = 0;

    if (!led1State && blinkCount > 0) {
        unsigned long currentMillis = millis();
        int blinkRate = max(50, 1000 / blinkCount);  // Min delay of 50ms

        if (currentMillis - previousMillis >= blinkRate) {
            previousMillis = currentMillis;
            led2State = !led2State;  // Toggle LED state
```

```
            led2.toggle();
        }
    } else {
        led2.turnOff();  // Ensure LED stays off if blinkCount is 0
    }
}



// Task 3: Variable State (Increment/Decrement)
void taskVariableState() {
    if (button2.isPressed()) {
        blinkCount++;  // Increment blink count
    } else if (button3.isPressed()) {
        blinkCount--;  // Decrement blink count
    }
}

// Idle Task: Reporting System State
void taskIdle() {
    // Reporting current system state via serial monitor
    printf("LED 1 State: %s\n", led1State ? "ON" : "OFF");
    printf("LED 2 Blink Count: %d\n", blinkCount);
    delay(200);  // Delay to avoid flooding the serial monitor
}

void setup() {
    // Initialize Serial communication
    serialInit();

    // Initialize LEDs to be turned off initially
    led1.turnOff();
    led2.turnOff();
}

void loop() {
    // Call the tasks in sequence
    taskButtonLED();         // Task 1: Button LED
    taskLEDIntermittent();   // Task 2: LED Intermittent
    taskVariableState();     // Task 3: Variable State
    taskIdle();              // Idle Task: Reporting state
}

#include "SerialIO.h"

```

```
86. int serial_putchar(char c, FILE* f) {
87.     Serial.write(c);
88.     return c;
89. }
90. 
91. int serial_getchar(FILE* f) {
92.     while (!Serial.available());
93.     return Serial.read();
94. }
95. 
96. FILE serial_stdout;
97. 
98. void serialInit() {
99.     Serial.begin(115200);
100.          while (!Serial);
101. 
102.          fdev_setup_stream(&serial_stdout, serial_putchar,
    serial_getchar, _FDEV_SETUP_RW);
103.          stdout = &serial_stdout;
104.          stdin = &serial_stdout;
105.      }
106. 
107.      #include "Led.h"
108. 
109.      Led::Led(uint8_t pin) {
110.          this->pin = pin;
111.          this->state = false;
112.          pinMode(pin, OUTPUT);
113.      }
114. 
115.      void Led::toggle() {
116.          state = !state;
117.          digitalWrite(pin, state ? HIGH : LOW);
118.      }
119. 
120.      void Led::turnOn() {
121.          state = true;
122.          digitalWrite(pin, HIGH);
123.      }
124. 
125.      void Led::turnOff() {
126.          state = false;
127.          digitalWrite(pin, LOW);
128.      }
129. 
```

```
#include "Button.h"

Button::Button(uint8_t pin) {
    this->pin = pin;
    pinMode(pin, INPUT_PULLUP);
}

bool Button::isPressed() {
    return digitalRead(pin) == HIGH;
}

```