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

Elaborated: Berco Andrei, 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

As embedded systems continue to play a vital role across various industries, microcontrollers (MCUs) have become indispensable for efficient human-machine interaction. Modern embedded applications often incorporate input devices like keypads and output displays such as LCDs to enhance user interfacing. Secure access mechanisms, including password-protected systems, are critical in areas such as security, automation, and control systems.

This laboratory work focuses on developing an understanding of peripheral interaction using the STDIO library to enable communication between a keypad and an LCD. The project involves designing an MCU-based system that verifies user-inputted codes and provides real-time visual feedback through LEDs and an LCD display.

2. Overview of the Hardware and Software Components Used

Hardware Components

The following hardware components are essential for implementing the sequential task execution and inter-task communication required in this laboratory work:

- **Microcontroller (Arduino Uno)** – Serves as the central processing unit, executing scheduled tasks and handling peripheral interactions.
- **Buttons (Minimum 3)** – Used for user input to trigger LED state changes and modify the recurrence variable.
- **LEDs** – Indicate system states, including button press detection and task execution conditions.
- **220 Ω Resistors** – Limit current to protect LEDs and other components.
- **Breadboard and Jumper Wires** – Facilitate prototyping and interconnection of components.

- **Power Supply (USB)** – Provides the necessary power for the microcontroller and connected peripherals.

Software Components

To implement the sequential task execution and inter-task communication, the following software tools and libraries are utilized:

- **Visual Studio Code with PlatformIO / Arduino IDE / ESP-IDF** – Development environments for writing, compiling, and uploading code to the microcontroller.
- **STDIO Library** – Used for serial communication, debugging, and reporting system states via printf().
- **Task Scheduling and Synchronization Methods** – Ensure non-preemptive execution and proper coordination between tasks.
- **Peripheral Control Libraries** – Enable interaction with hardware components such as buttons, LEDs, and displays as required by the application.

3. System Architecture Explanation and Solution Justification

The system is designed using a modular approach, where each task operates independently while following a sequential execution model. The microcontroller continuously monitors user input through buttons, executing specific tasks based on the detected signals. Task execution follows a structured schedule, ensuring efficient resource utilization and proper synchronization between components.

- **Task 1: Button LED** – Toggles an LED state upon button press detection.
- **Task 2: LED Intermitent** – Controls a blinking LED, active only when the first LED is off.
- **Task 3: State Variable** – Adjusts a recurrence variable based on button inputs, influencing Task 2’s behavior.
- **Idle Task** – Collects and reports system state information using printf(), demonstrating provider-consumer communication.

This approach ensures efficient, predictable, and well-structured task execution, making it applicable to embedded control systems requiring precise timing and resource management.

Why This Solution?

- **Reliability** – Ensures consistent execution and synchronization between tasks.
- **Scalability** – Can be extended with additional tasks, peripherals, or wireless communication.
- **Ease of Implementation** – Uses widely available microcontrollers, buttons, and LEDs.
- **Optimized Resource Utilization** – Reduces processor load through efficient scheduling techniques.

This system effectively demonstrates sequential task execution, synchronization, and real-world embedded system principles.

4. Case Study: Sequential Task Execution in Embedded Systems

Context and Necessity

In embedded systems, efficient task scheduling is crucial for ensuring reliable execution of multiple operations without preemptive multitasking. Industrial automation, home security, and resource monitoring systems often rely on microcontroller-based solutions that follow sequential execution principles. This case study examines a modular approach for managing LED control and button interactions using a non-preemptive scheduling model on an MCU.

Practical Implementation

A microcontroller (such as Arduino or ESP32) is programmed to handle multiple tasks sequentially:

- **Task 1: Button LED Control** – Detects a button press and toggles an LED state.
- **Task 2: Intermittent LED** – Controls a blinking LED, active only when the first LED is off.

- **Task 3: State Variable Adjustment** – Modifies a variable using two buttons, influencing the LED blinking frequency.
- **Idle Task: System Monitoring** – Uses printf() via STDIO to report task states, demonstrating provider-consumer synchronization.

Each LED in the system has a specific role:

- **Primary LED** – Toggles on button press.
- **Intermittent LED** – Blinks based on the state variable value.

Example Workflow:

1. A user presses a button, toggling the primary LED state.
2. If the primary LED is off, the intermittent LED starts blinking at a rate defined by the state variable.
3. Pressing two additional buttons increases or decreases the blinking rate of the intermittent LED.
4. The Idle Task continuously monitors and logs system state changes.

Extending the Case Study

This methodology can be adapted for various embedded system applications:

- **Energy-Efficient Scheduling** – Optimizing task timing to reduce CPU load.
- **Event-Driven Automation** – Triggering task execution based on sensor inputs.
- **Remote Monitoring** – Logging system states via wireless communication modules.

Benefits and Impact

Using sequential task execution in microcontroller-based systems offers several advantages:

- **Predictability** – Ensures tasks run in a structured and synchronized manner.
- **Resource Optimization** – Reduces processor load by avoiding unnecessary interrupts.
- **Modularity** – Enables easy expansion with additional tasks and functionalities.

- **Scalability** – Can be integrated with wireless communication, IoT frameworks, or real-time monitoring systems.

This case study highlights the importance of structured scheduling in embedded applications, demonstrating how sequential execution models enhance reliability and efficiency in task management.

Design

1. Architectural Sketch and Component Interconnection

The system architecture is based on the interaction between:

- An **Arduino Mega 2560** microcontroller,
- A **PC** for sending serial commands,
- 2 **LEDs** connected to a digital pin of the microcontroller.
- 3 buttons
- Wires and resistors

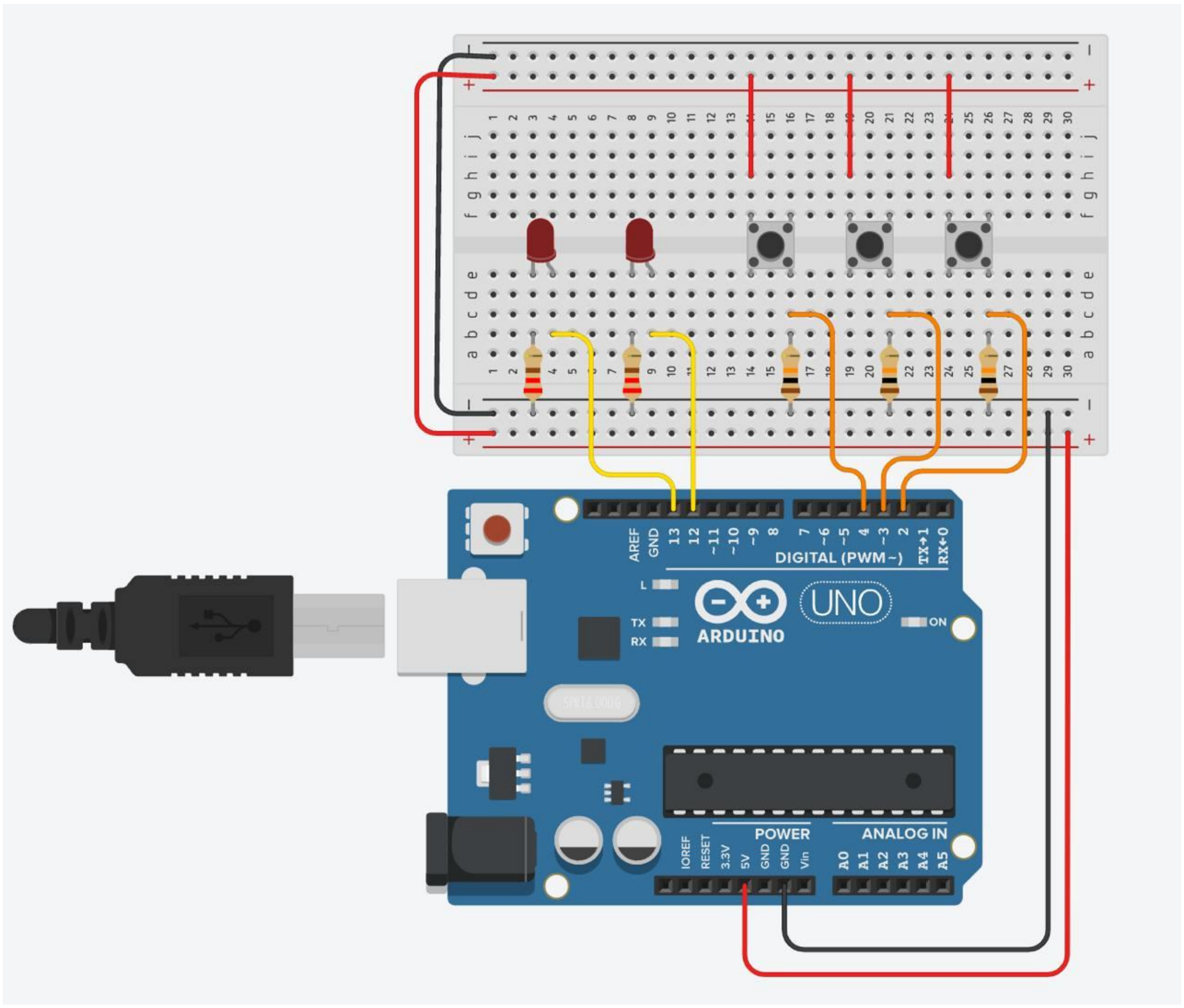


Figure 2.1 Component scheme

1. **Ground and Power Connections:**
   - The ground (GND) pin of the Arduino is connected to the ground rail of the breadboard.
   - The 5V pin of the Arduino is connected to the power rail of the breadboard.
2. **LED Connections:**
   - Digital pins 8, 9, and 10 on the Arduino are each connected to one end of a resistor on the breadboard.
   - Each resistor is then connected in series with an LED, with the other end of each LED going to ground.
3. **Button Connections:**

    - Place three buttons on the breadboard.
    - Connect one terminal of each button to the ground rail of the breadboard.
    - Connect the other terminal of each button to digital pins 2, 3, and 4 on the Arduino.
    - Additionally, connect a pull-up resistor (10k ohms) between each button's terminal (connected to the digital pins) and the 5V rail. This ensures that the input pin reads HIGH when the button is not pressed.

Here's a summary of the connections:

- **LEDs:**
    - LED 1: Digital pin 8 -> Resistor -> LED -> Ground
    - LED 2: Digital pin 9 -> Resistor -> LED -> Ground
    - LED 3: Digital pin 10 -> Resistor -> LED -> Ground
- **Buttons:**
    - Button 1: Digital pin 2 -> Button -> Ground (with pull-up resistor to 5V)
    - Button 2: Digital pin 3 -> Button -> Ground (with pull-up resistor to 5V)
    - Button 3: Digital pin 4 -> Button -> Ground (with pull-up resistor to 5V)

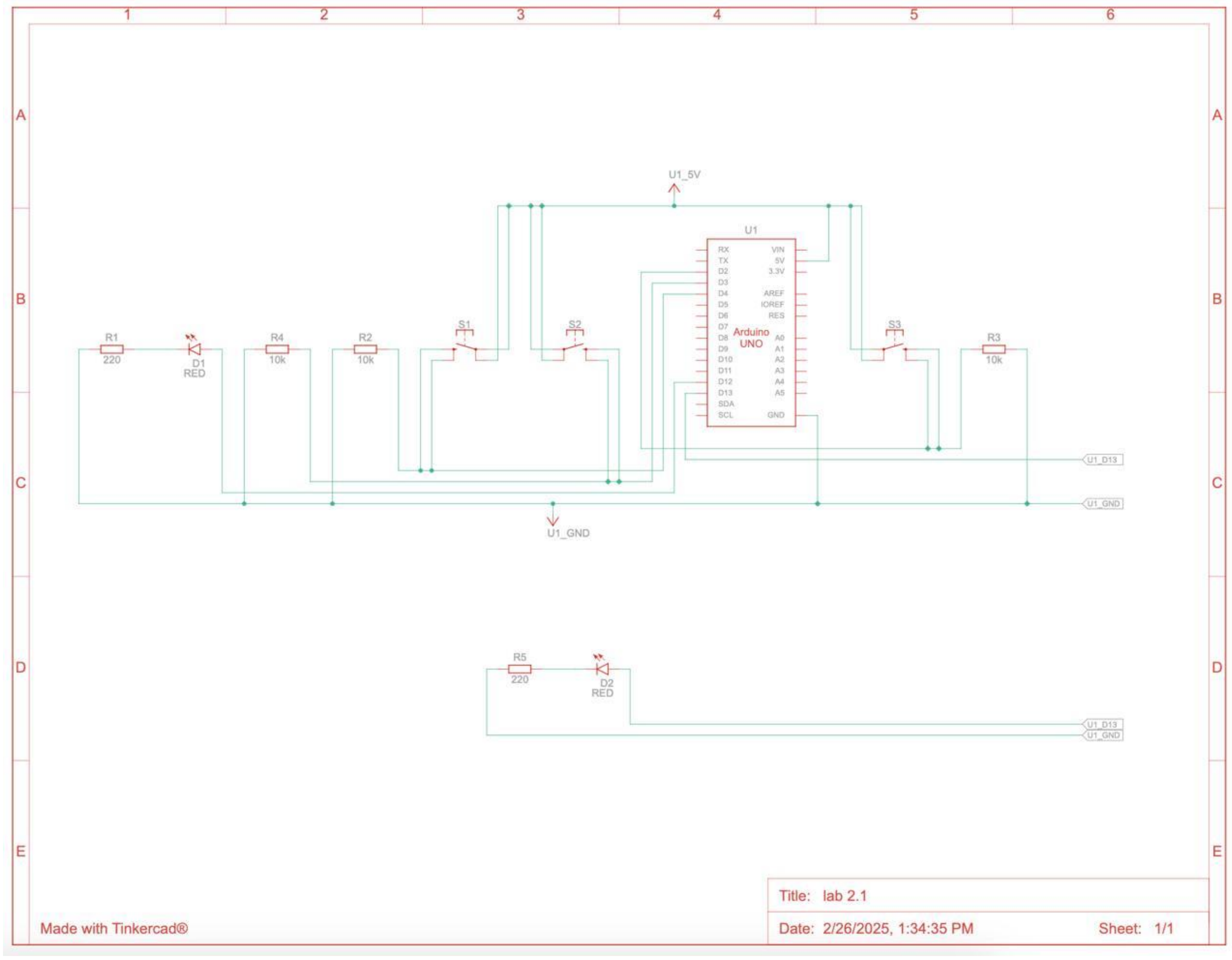


Figure 2.2 Electrical scheme

1. **Arduino Uno:**
    - The Arduino Uno is the central component, with various pins connected to other elements in the circuit.
2. **Power and Ground Connections:**
    - The 5V pin of the Arduino is connected to the power rail of the breadboard.
    - The GND pin of the Arduino is connected to the ground rail of the breadboard.
3. **LED Connections:**
    - **LED 1:** Connected to digital pin 8 through a resistor.
    - **LED 2:** Connected to digital pin 9 through a resistor.
    - **LED 3:** Connected to digital pin 10 through a resistor.

- Each LED is connected in series with a resistor, and the other end of each LED is connected to the ground rail.

4. **Button Connections:**
   - **Button 1:** Connected to digital pin 2 and ground, with a pull-up resistor to 5V.
   - **Button 2:** Connected to digital pin 3 and ground, with a pull-up resistor to 5V.
   - **Button 3:** Connected to digital pin 4 and ground, with a pull-up resistor to 5V.
5. **Resistors:**
   - Resistors are used in series with the LEDs to limit the current.
   - Pull-up resistors (10k ohms) are used with the buttons to ensure the input pins read HIGH when the buttons are not pressed.
6. **Capacitors:**
   - There are no capacitors shown in this particular schematic.

2. Scheme bloc and algorithm

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

Flowchart – Serial Command Processing
(A Flowchart diagram illustrating the cycle: command reception → processing → execution → user feedback)

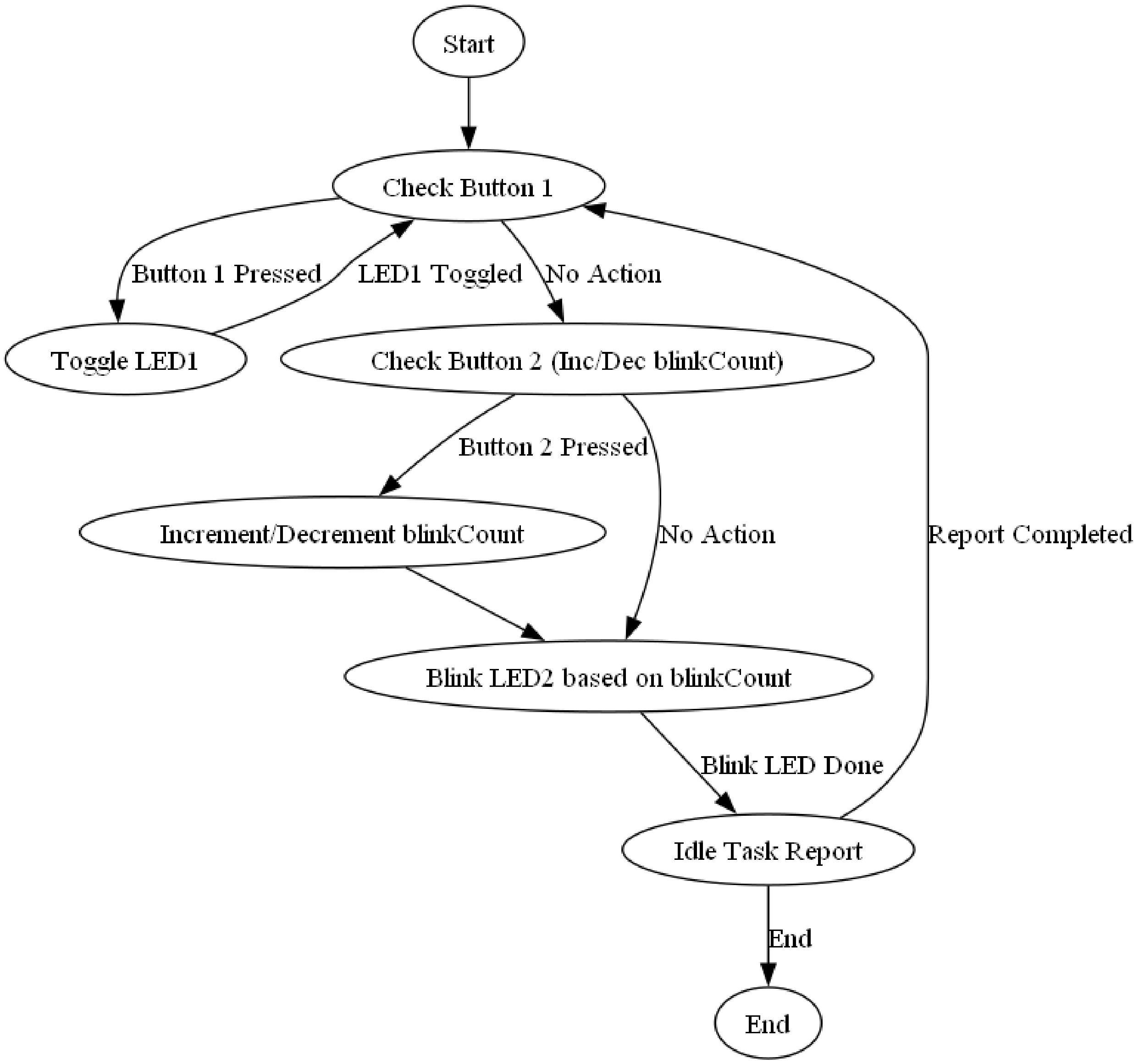


Figure 2.3 Flowchart

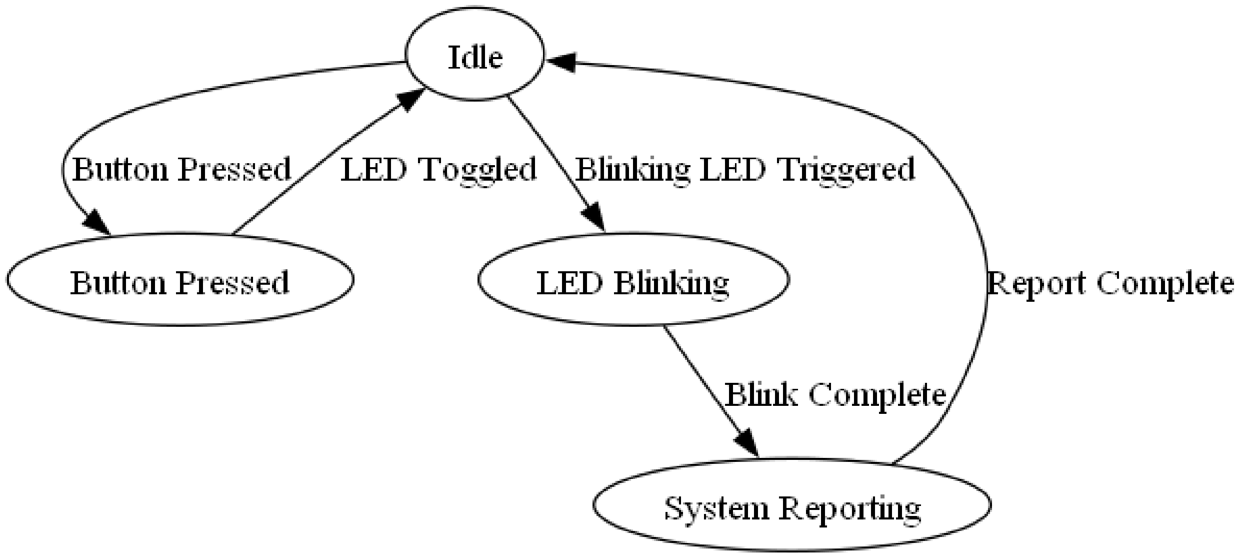
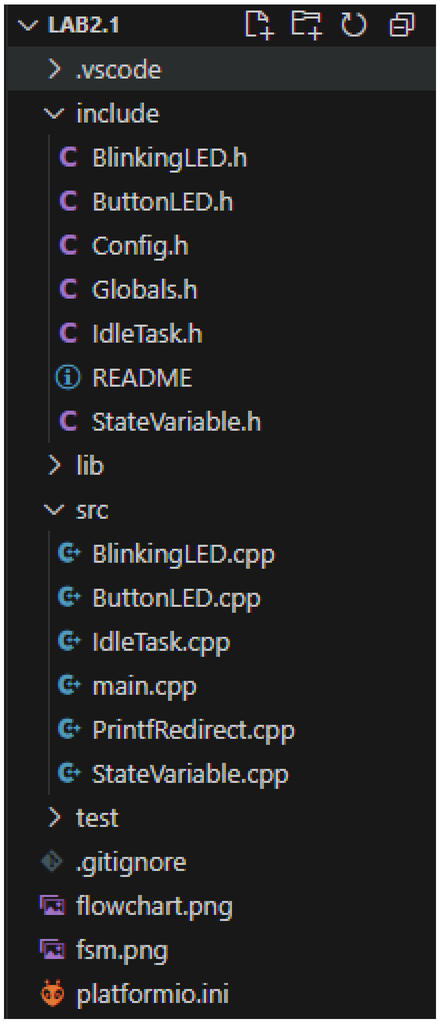


Figure 2.4 FSM diagram

3. Modular implementation

For better project organization, a modular architecture was used, dividing functionalities into separate files.



```
LAB2.1
  > .vscode
  v include
      BlinkingLED.h
      ButtonLED.h
      Config.h
      Globals.h
      IdleTask.h
      README
      StateVariable.h
  > lib
  v src
      BlinkingLED.cpp
      ButtonLED.cpp
      IdleTask.cpp
      main.cpp
      PrintfRedirect.cpp
      StateVariable.cpp
  > test
    .gitignore
    flowchart.png
    fsm.png
    platformio.ini
```

Figure 3.1 Project otionrganization

The project is well-structured, separating functionality into different modules using a layered approach. The organization follows best practices for embedded systems and modular programming.

Code Functionality Overview:

1. **Configuration Module (`Config.h`)**
   - Defines constants and pin assignments for buttons and LED components.
   - Provides a predefined valid code for authentication.
2. **BlinkingLED.cpp**

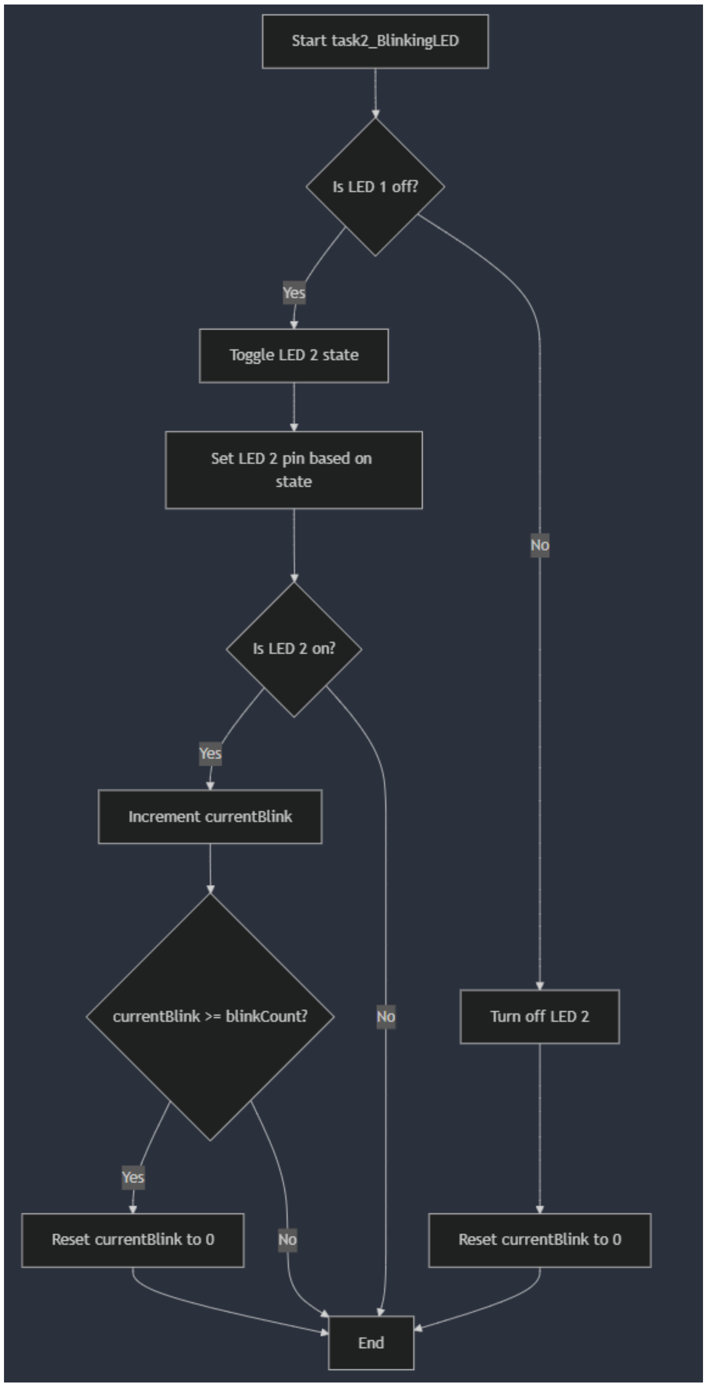


Figure 3.2: Flowchart forBlinkingLed.cpp

This file manages LED 2's blinking behavior. It blinks LED 2 only when LED 1 is off. The blinking follows a pattern where it counts up to a user-defined number of blinks (blinkCount), then resets. When LED 1 is turned on, LED 2 is turned off and the blink counter resets.

3. **ButtonLED.cpp**

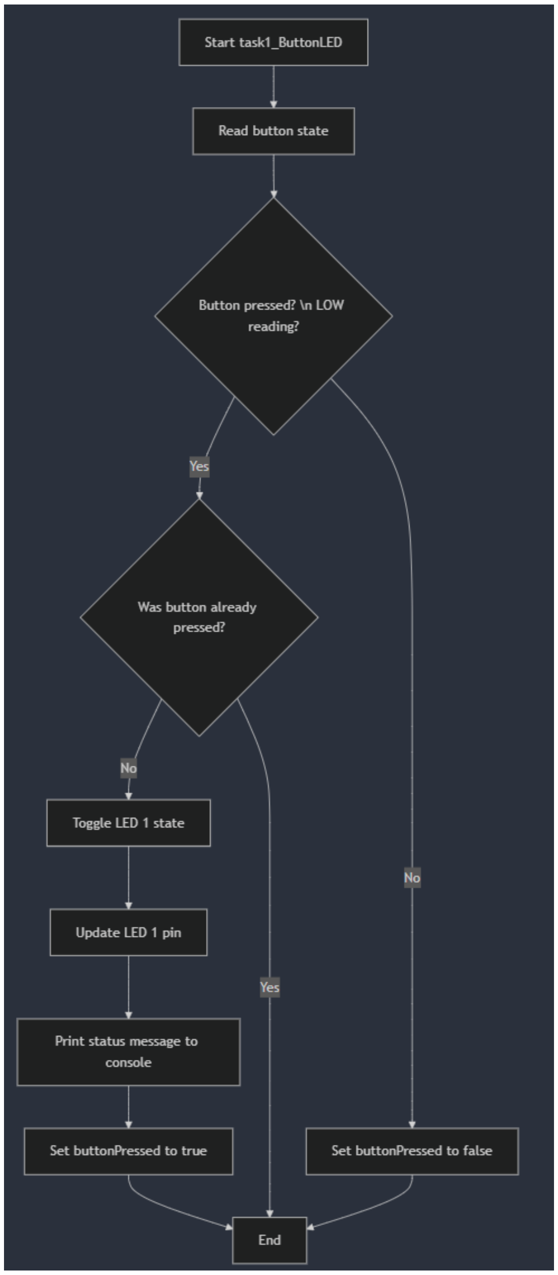

Start task1_ButtonLED
Read button state
Button pressed? \n LOW reading?
Yes
Was button already pressed?
No
Toggle LED 1 state
Update LED 1 pin
Print status message to console
Set buttonPressed to true
Yes
No
Set buttonPressed to false
End

Figure 3.3: Flowchart for ButtonLED.cpp

This file controls LED 1 based on button input. When Button 1 is pressed (LOW reading), it toggles LED 1 between on and off states. The code includes a button debouncing mechanism so the LED only toggles once per button press. It also outputs the LED state to the serial console.

4. **IdleTask.cpp**

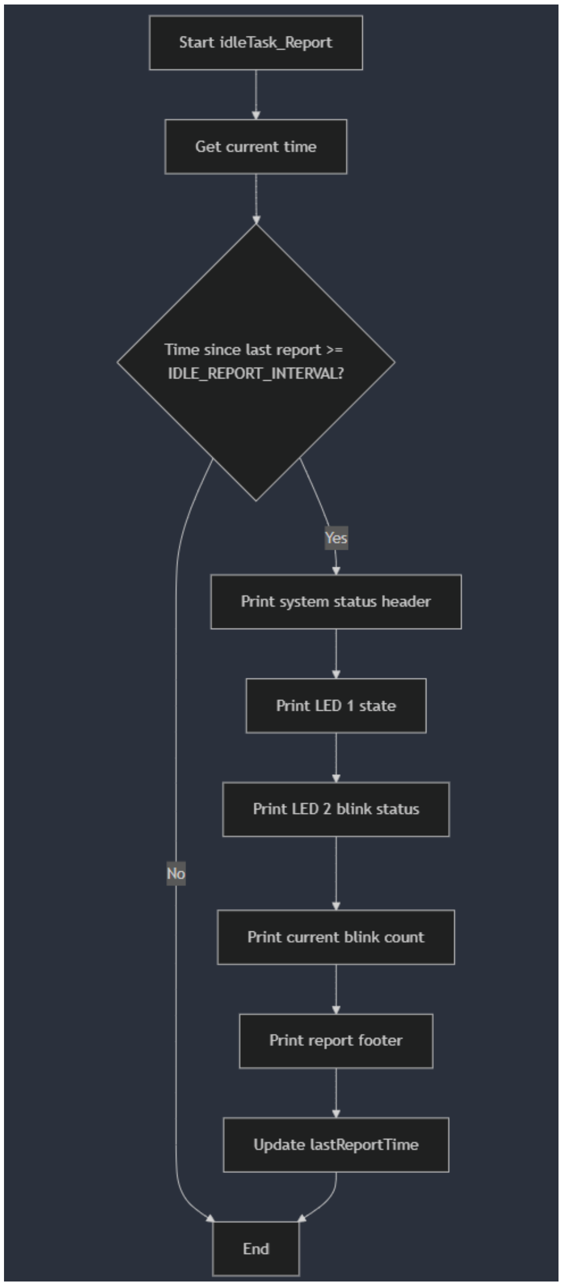


Figure 3.4: Flowchart for IdleTask.cpp

This file implements a reporting mechanism that runs during idle time. At regular intervals (defined by IDLE_REPORT_INTERVAL), it outputs a system status report to the serial console. The report includes the state of LED 1, whether LED 2 blinking is active, and the current blink count setting.

5. **StateVariable.cpp**

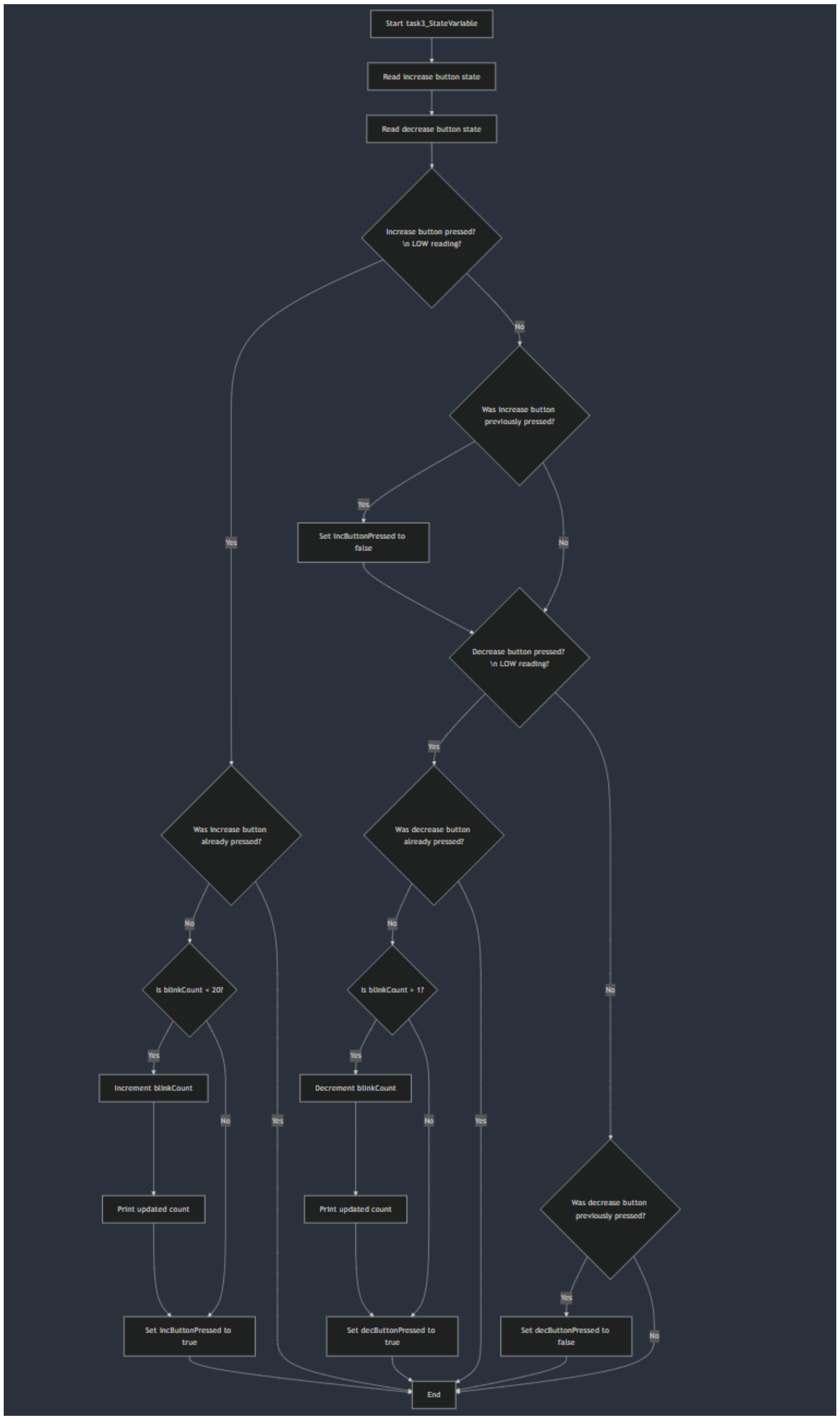

Start task3_StateVariable
Read increase button state
Read decrease button state
Increase button pressed? \n LOW reading?
Yes
No
Was increase button previously pressed?
Yes
Set incButtonPressed to false
No
Decrease button pressed? \n LOW reading?
Yes
No
Was increase button already pressed?
No
Yes
Is blinkCount < 20?
Yes
No
Increment blinkCount
Print updated count
Set incButtonPressed to true
Was decrease button already pressed?
No
Yes
Is blinkCount > 1?
Yes
No
Decrement blinkCount
Print updated count
Set decButtonPressed to true
Was decrease button previously pressed?
Yes
No
Set decButtonPressed to false
End

Figure 3.5: Flowchart for StateVariable.cpp

This file manages the blinkCount variable, which determines how many times LED 2 blinks before resetting. It reads from two buttons: one to increase the count (up to a maximum of 20) and one to decrease it (down to a minimum of 1). Button debouncing is implemented to ensure a single button press registers only once. Changes to the blink count are reported to the serial console.

6. main.cpp

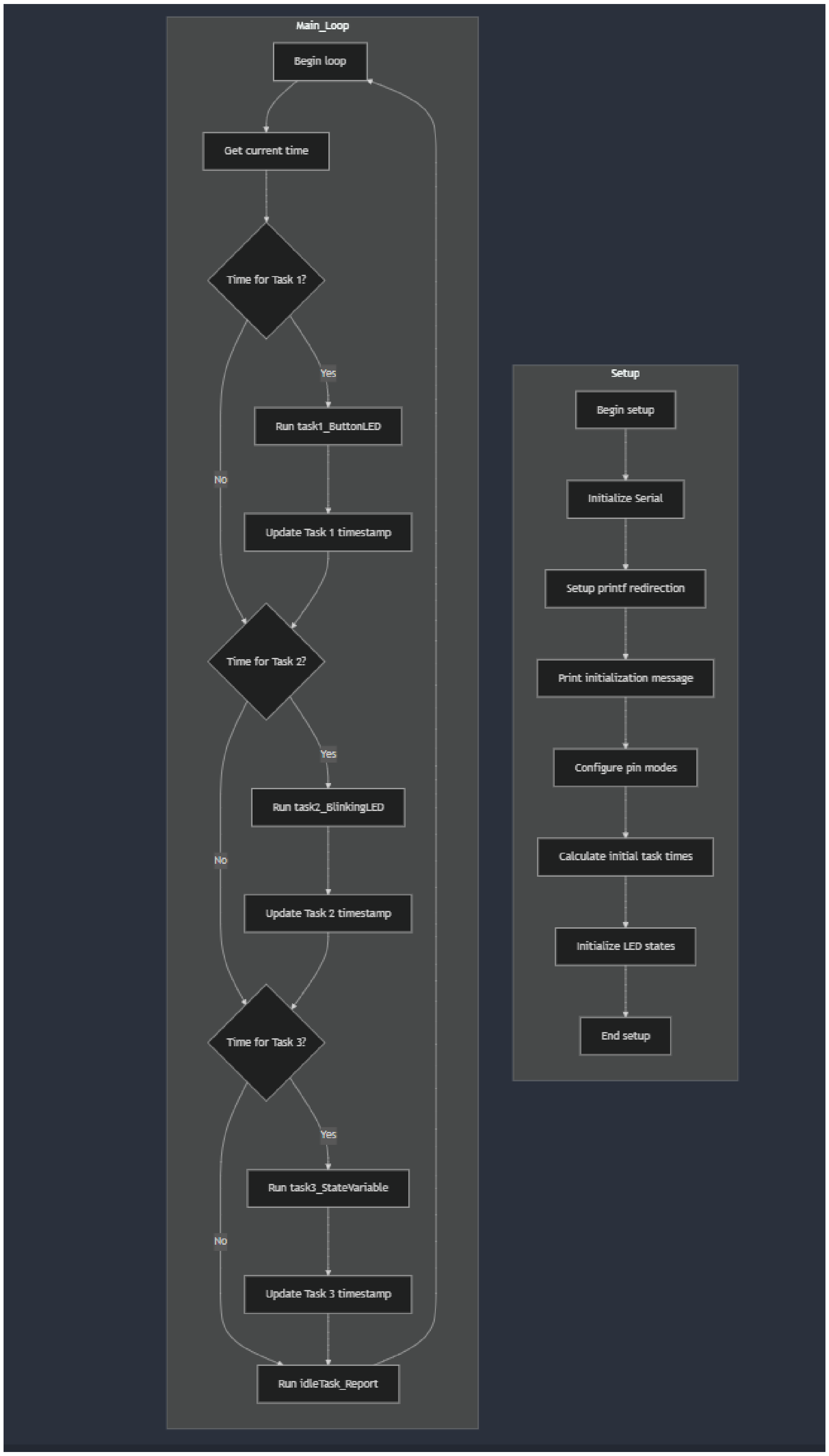


Figure 3.5: Flowchart for StateVariable.cpp

This is the main program file that initializes the Arduino system and implements a simple cooperative multitasking system. It sets up pins, initializes variables, and contains the main loop that schedules three tasks at different recurrence intervals. The tasks manage button input, LED blinking, and state variable management. An idle task reports system status when no other tasks are running.

Results

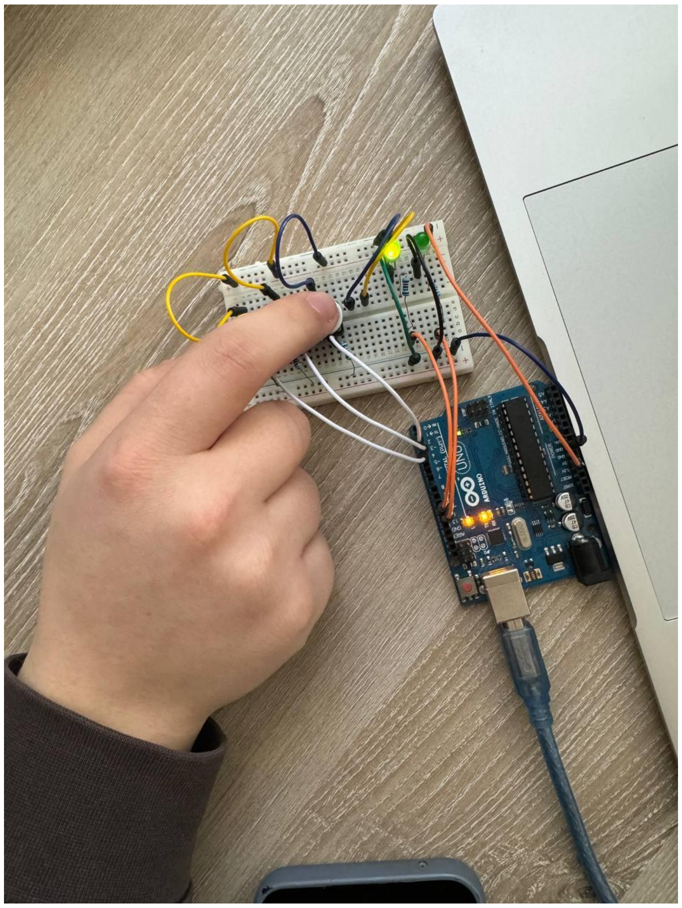


Figure 4.1 Button pressed for static LED

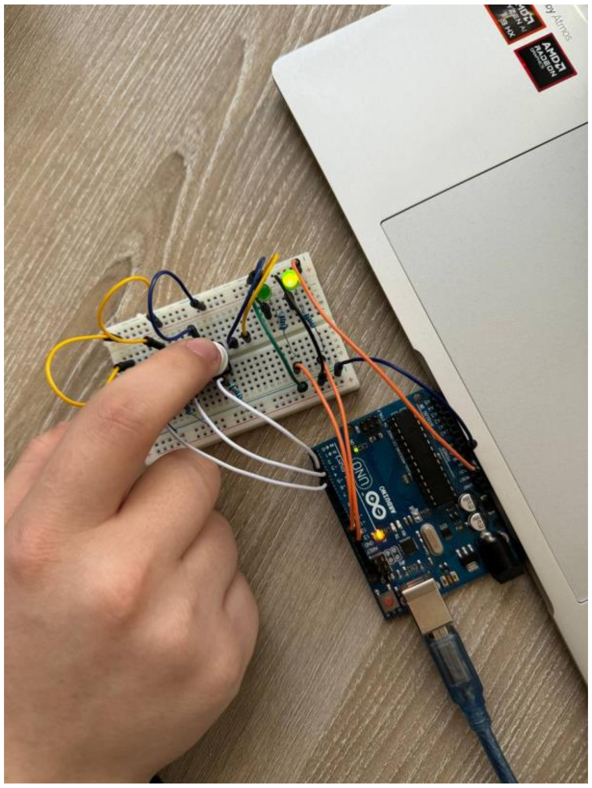
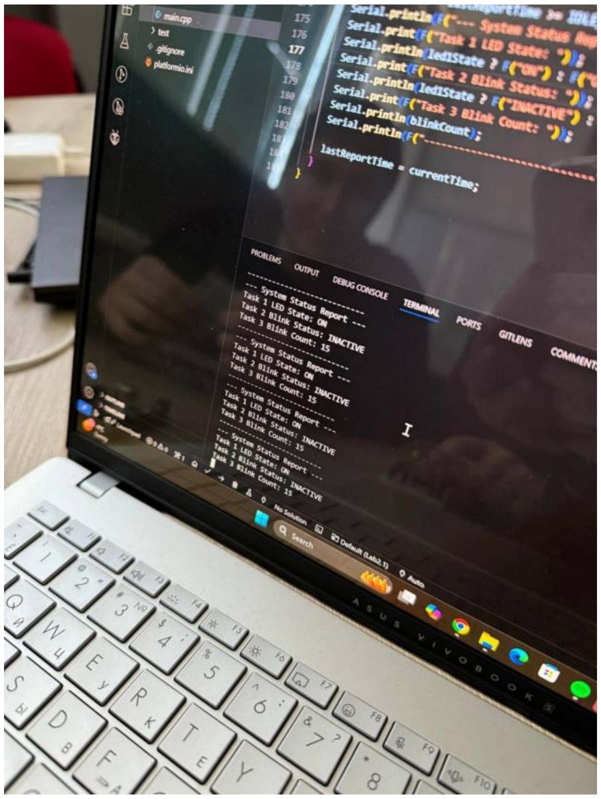


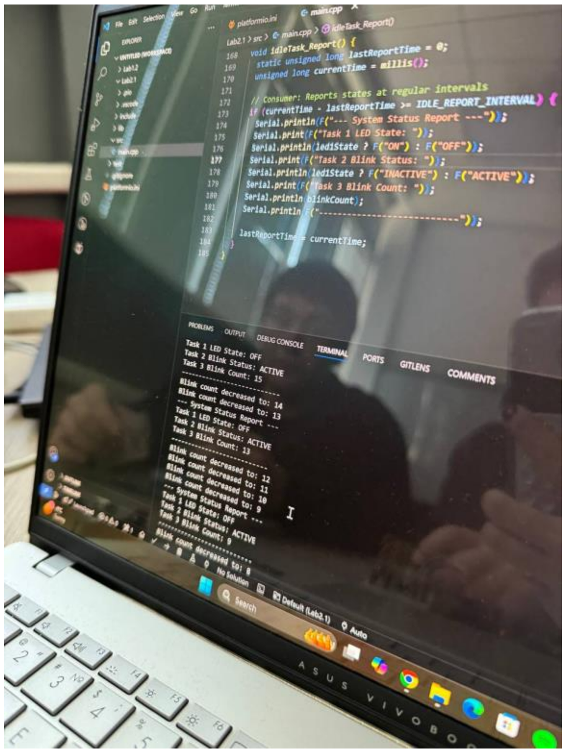
Figure 4.2 Button pressed for linking LED



```
Serial.println(F("--- System Status Rep
Serial.print(F("Task 1 LED State: "));
Serial.println(led1State ? F("ON") : F("
Serial.print(F("Task 2 Blink Status: "));
Serial.println(led1State ? F("INACTIVE") :
Serial.print(F("Task 3 Blink Count: "));
Serial.println(blinkCount);
Serial.println(F("------------------------------
lastReportTime = currentTime;
}
}
```

```
--------------------------
--- System Status Report ---
Task 1 LED State: ON
Task 2 Blink Status: INACTIVE
Task 3 Blink Count: 15
--------------------------
--- System Status Report ---
Task 1 LED State: ON
Task 2 Blink Status: INACTIVE
Task 3 Blink Count: 15
--------------------------
--- System Status Report ---
Task 1 LED State: ON
Task 2 Blink Status: INACTIVE
Task 3 Blink Count: 15
--------------------------
--- System Status Report ---
Task 1 LED State: ON
Task 2 Blink Status: INACTIVE
Task 3 Blink Count: 15
```

Figure 4.3 System report



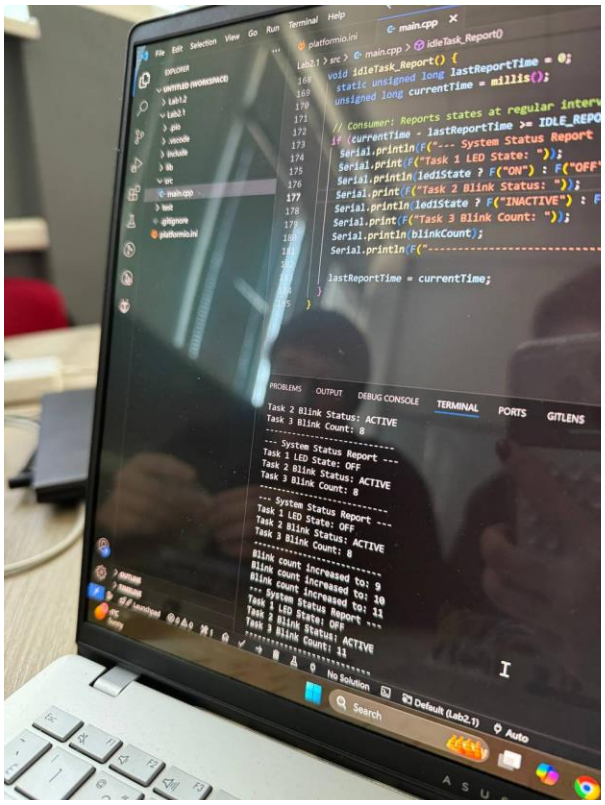
```cpp
void idleTask_Report() {
  static unsigned long lastReportTime = 0;
  unsigned long currentTime = millis();

  // Consumer: Reports states at regular intervals
  if (currentTime - lastReportTime >= IDLE_REPORT_INTERVAL) {
    Serial.println(F("--- System Status Report ---"));
    Serial.print(F("Task 1 LED State: "));
    Serial.println(led1State ? F("ON") : F("OFF"));
    Serial.print(F("Task 2 Blink Status: "));
    Serial.println(led1State ? F("INACTIVE") : F("ACTIVE"));
    Serial.print(F("Task 3 Blink Count: "));
    Serial.println(blinkCount);
    Serial.println(F("----------------------------"));

    lastReportTime = currentTime;
  }
}
```

```
Task 1 LED State: OFF
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 15
----------------------------
Blink count decreased to: 14
Blink count decreased to: 13
--- System Status Report ---
Task 1 LED State: OFF
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 13
----------------------------
Blink count decreased to: 12
Blink count decreased to: 11
Blink count decreased to: 10
Blink count decreased to: 9
--- System Status Report ---
Task 1 LED State: OFF
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 9
----------------------------
Blink count decreased to: 8
```

Figure 4.4 System report after pressing the button for decreasing



```cpp
void idleTask_Report() {
  static unsigned long lastReportTime = 0;
  unsigned long currentTime = millis();

  // Consumer: Reports states at regular interv
  if (currentTime - lastReportTime >= IDLE_REPO
    Serial.println(F("--- System Status Report
    Serial.print(F("Task 1 LED State: "));
    Serial.println(led1State ? F("ON") : F("OFF"
    Serial.print(F("Task 2 Blink Status: "));
    Serial.println(led1State ? F("INACTIVE") : F
    Serial.print(F("Task 3 Blink Count: "));
    Serial.println(blinkCount);
    Serial.println(F("--------------------------

    lastReportTime = currentTime;
  }
}
```

```
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 8
----------------------------
--- System Status Report ---
Task 1 LED State: OFF
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 8
----------------------------
--- System Status Report ---
Task 1 LED State: OFF
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 8
----------------------------
Blink count increased to: 9
Blink count increased to: 10
Blink count increased to: 11
--- System Status Report ---
Task 1 LED State: OFF
Task 2 Blink Status: ACTIVE
Task 3 Blink Count: 11
----------------------------
```

Figure 4.5 System report after pressing the button for increasing

Conclusions

After completing this laboratory work, the following results were achieved:

A functional system was successfully implemented, enabling interaction between various components, including LEDs, buttons, and the state variable system using modular tasks.

- The application correctly handles input from buttons to toggle LED states, adjust the blink count, and report system status through the serial interface.
- A modular architecture was employed, ensuring clear separation of functionalities for peripheral control (LED blinking, button handling, and state variable management).
- The system was tested on a microcontroller platform (such as Arduino Uno or ESP32) and validated using a simulator to confirm its correctness.
- A visual indicator (LED) was integrated to provide immediate feedback—green for a valid button press or state change and red for invalid actions or states.

System Performance Analysis

The system demonstrated stable and efficient operation, with reliable handling of button presses, LED blinking tasks, and periodic status reporting. Key performance aspects include:

- **Response Time**: Button presses and state changes are processed almost instantly, with minimal delay in the LED toggling and blink count adjustments.
- **Reliability**: The application consistently detects user input, toggles LEDs, adjusts the blink count, and reports the system status accurately.
- **Modularity**: The structured design allows for easy extension, such as adding more peripheral control tasks or modifying the LED blinking logic.

Identified Limitations

Despite the system's functionality, several areas for improvement were identified:

1. **Button Press Handling** – The system currently toggles LEDs with basic button presses, but more sophisticated user feedback (such as debouncing) could be implemented for improved reliability.
2. **Limited Task Complexity** – The system handles basic tasks, but more advanced interactions (like multiple LED control or task prioritization) could enhance the complexity and functionality.
3. **Fixed Blink Count** – The blink count is predefined, limiting flexibility. Implementing a dynamic or user-configurable blink count would make the system more versatile.

Impact of the Technology Used in Real-World Applications

This project serves as a practical example of embedded systems with modular task handling and user interaction, offering insights into the following real-world applications:

• **Embedded Systems Design**: The project demonstrates how to structure code for modularity, making it suitable for embedded systems with similar user interactions.

• **Home Automation**: The concept of toggling LEDs and adjusting settings based on button presses can be applied to smart home devices and systems.

• **Security Systems**: The button-controlled state transitions and LED indicators are applicable in security systems for user authentication or status indication.

Improvement Suggestions

To enhance the project, the following improvements could be considered:

1. **Enhanced Button Handling** – Implementing debouncing mechanisms to ensure reliable button press detection.
2. **Advanced Task Handling** – Adding task prioritization and more complex interactions between tasks could improve the system's flexibility.

3. **Dynamic Blink Count Configuration** – Allowing users to set or change the blink count dynamically instead of relying on a fixed value would increase system versatility.
4. **Error Handling** – Improving the handling of invalid button presses or tasks would make the system more robust.
5. **Wireless Integration** – Using an ESP32 module to enable wireless control or monitoring of the LED states through Wi-Fi or Bluetooth.

By implementing these optimizations, the system could be made more robust, user-friendly, and adaptable to a wider range of applications, providing a more flexible and reliable solution for real-world embedded system scenarios.

Note on AI Tool Usage

During the drafting of this report, the author used ChatGPT for generating and structuring the content. The resulting information was reviewed, validated, and adjusted according to the requirements of the laboratory work, ensuring technical accuracy and clarity of explanations. The use of this AI tool was aimed at structuring and optimizing the presentation of information without replacing personal analysis and understanding of the subject.

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

Appendix

1. **GitHub**: https://github.com/KaBoomKaBoom/ES_Labs.git
2. **YouTube Demo:** https://youtube.com/shorts/91sfxRM22Ys
3. **Config.h**

```
#ifndef CONFIG_H
#define CONFIG_H

// Pin definitions
const int BUTTON_1_PIN = 2;
const int LED_1_PIN = 11;
const int LED_2_PIN = 12;
const int INC_BUTTON_PIN = 3;
const int DEC_BUTTON_PIN = 4;

// Task recurrence times
const unsigned long TASK1_RECURRENCE = 50;
const unsigned long TASK2_RECURRENCE = 100;
const unsigned long TASK3_RECURRENCE = 100;
const unsigned long IDLE_REPORT_INTERVAL = 1000;

// Task offsets
const unsigned long TASK1_OFFSET = 0;
const unsigned long TASK2_OFFSET = 20;
const unsigned long TASK3_OFFSET = 40;

#endif
```

4. BlinkingLed.h

```
#ifndef BLINKING_LED_H
#define BLINKING_LED_H

void task2_BlinkingLED();

#endif
```

5. ButtonLed.h

```
#ifndef BUTTON_LED_H
#define BUTTON_LED_H

void task1_ButtonLED();

#endif
```

6. Globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

#include <Arduino.h>
#include "Config.h"

volatile bool led1State = false;
volatile int blinkCount = 5;
volatile unsigned long lastTaskTime[3];
volatile bool buttonPressed = false;
volatile bool incButtonPressed = false;
volatile bool decButtonPressed = false;

void setupPrintf();
```

```
#endif
```

7. IdleTask.h

```
#ifndef IDLE_TASK_H
#define IDLE_TASK_H

void idleTask_Report();

#endif
```

8. StateVariable.h

```
#ifndef STATE_VARIABLE_H
#define STATE_VARIABLE_H

void task3_StateVariable();

#endif
```

9. BlinkingLed.cpp

```
#include "BlinkingLED.h"
#include "Globals.h"

void task2_BlinkingLED() {
    static bool led2State = false;
    static int currentBlink = 0;

    if (!led1State) {
        led2State = !led2State;
        digitalWrite(LED_2_PIN, led2State ? HIGH : LOW);
        if (led2State) {
            currentBlink++;
            if (currentBlink >= blinkCount) {
                currentBlink = 0;
            }
        }
    } else {
        digitalWrite(LED_2_PIN, LOW);
        currentBlink = 0;
    }
}
```

10. ButtonLedmain.cpp

```
#include "ButtonLED.h"
#include "Globals.h"

void task1_ButtonLED() {
    int buttonReading = digitalRead(BUTTON_1_PIN);

    if (buttonReading == LOW) {
        if (!buttonPressed) {
            led1State = !led1State;
            digitalWrite(LED_1_PIN, led1State ? HIGH : LOW);
            printf("Button pressed! LED1 is now: %s\n", led1State ? "ON" : "OFF");
            buttonPressed = true;
        }
    } else {
        buttonPressed = false;
    }
}
```

11. IdleTask.cpp

```
#include "IdleTask.h"
#include "Globals.h"

void idleTask_Report() {
    static unsigned long lastReportTime = 0;
    unsigned long currentTime = millis();

    if (currentTime - lastReportTime >= IDLE_REPORT_INTERVAL) {
        printf("--- System Status Report ---\n");
        printf("Task 1 LED State: %s\n", led1State ? "ON" : "OFF");
        printf("Task 2 Blink Status: %s\n", led1State ? "INACTIVE" : "ACTIVE");
        printf("Task 3 Blink Count: %d\n", blinkCount);
        printf("---------------------------\n");

        lastReportTime = currentTime;
    }
}
```

12. PrintfRedirect.cpp

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

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

void setupPrintf() {
    static FILE serial_stdout;
    fdev_setup_stream(&serial_stdout, serialPutchar, NULL, _FDEV_SETUP_WRITE);
    stdout = &serial_stdout;
  }
```

13. StateVariable.cpp

```
#include "StateVariable.h"
#include "Globals.h"

void task3_StateVariable() {
  bool incButtonState = digitalRead(INC_BUTTON_PIN) == LOW;
  bool decButtonState = digitalRead(DEC_BUTTON_PIN) == LOW;

  if (incButtonState && !incButtonPressed) {
    if (blinkCount < 20) {
      blinkCount++;
      printf("Blink count increased to: %d\n", blinkCount);
    }
    incButtonPressed = true;
  } else if (!incButtonState && incButtonPressed) {
    incButtonPressed = false;
  }

  if (decButtonState && !decButtonPressed) {
    if (blinkCount > 1) {
      blinkCount--;
      printf("Blink count decreased to: %d\n", blinkCount);
    }
    decButtonPressed = true;
  } else if (!decButtonState && decButtonPressed) {
    decButtonPressed = false;
  }
}
```

14. main.cpp

```
#include <Arduino.h>
#include "ButtonLED.h"
#include "BlinkingLED.h"
#include "StateVariable.h"
#include "IdleTask.h"
#include "Globals.h"

void setup() {
    Serial.begin(9600);
    setupPrintf();

    printf("Sequential Task System Initialized\n");

    pinMode(BUTTON_1_PIN, INPUT_PULLUP);
    pinMode(LED_1_PIN, OUTPUT);
    pinMode(LED_2_PIN, OUTPUT);
    pinMode(INC_BUTTON_PIN, INPUT_PULLUP);
    pinMode(DEC_BUTTON_PIN, INPUT_PULLUP);

    unsigned long currentTime = millis();
    lastTaskTime[0] = currentTime + TASK1_OFFSET;
    lastTaskTime[1] = currentTime + TASK2_OFFSET;
    lastTaskTime[2] = currentTime + TASK3_OFFSET;

    digitalWrite(LED_1_PIN, led1State);
    digitalWrite(LED_2_PIN, LOW);
}

void loop() {
    unsigned long currentTime = millis();

    if (currentTime - lastTaskTime[0] >= TASK1_RECURRENCE) {
      task1_ButtonLED();
      lastTaskTime[0] = currentTime;
    }

    if (currentTime - lastTaskTime[1] >= TASK2_RECURRENCE) {
      task2_BlinkingLED();
      lastTaskTime[1] = currentTime;
    }

    if (currentTime - lastTaskTime[2] >= TASK3_RECURRENCE) {
      task3_StateVariable();
      lastTaskTime[2] = currentTime;
    }

    idleTask_Report();
}
```