TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATION

Laboratory work 2.2

Subject: Operating Systems - FreeRTOS

Done by: **Konjevic Alexandra,**
st. gr. FAF-213

Verified by: **Moraru Dumitru,**
university lecturer

CHIŞINĂU, 2024

THE TASKS OF THE LABORATORY WORK

Creating an application for MCU (Microcontroller Unit) that will run at least 3 tasks using FreeRTOS.

This project aims to develop an application for a microcontroller (MCU) that executes at least three tasks sequentially. The tasks are as follows:

1. Button Led - LED state change upon detection of a button press.
2. A second Flashing LED in the phase when the LED from the first Task is off
3. Increment/decrement the value of a variable when pressing two buttons that will represent the number of recurrences/time during which the led from the second task will be in a state.
4. The Idle task will be used to display the states in the program, such as LED status display, and message display when the buttons are pressed, an implementation being that when the button is pressed, a variable is set, and when the message is displayed - reset, implementing the provider/consumer mechanism.

PROGRESS OF THE WORK

1.1 Description

This program is an example of a multitasking application for a microcontroller unit (MCU) using the FreeRTOS (Real-Time Operating System) library. FreeRTOS allows for the creation of multiple tasks that can run concurrently, enabling efficient utilization of the MCU's resources.

The program consists of several tasks:

1. Button Task (button_task): This task monitors the state of a button (LED_BUTTON). Also, it signals when the button is pressed using a semaphore (any_button_pressed) and signals UI update using another semaphore (update_ui).

2. Flicker Task (flicker_task): This task is responsible for controlling the flickering state of an LED (RED_LED) when the button is not pressed. It also adjusts flicker interval based on button presses and signals UI update upon button presses.

3. UI Task (idle_task): It handles LCD display and user interface and displays messages upon button presses. Also, this task updates UI elements based on semaphore signals.

4. UI LEDs Task (leds_control_task): Controls the state of LEDs (YELLOW_LED and RED_LED) based on button presses and flicker state.

The tasks communicate with each other using semaphores (any_button_pressed and update_ui). Semaphores are synchronization mechanisms used to signal events between tasks in a multitasking environment. In this program, semaphores are employed to notify tasks about button presses and UI updates. When a button is pressed, the corresponding semaphore is given, indicating to other tasks that a button event has occurred. Similarly, when UI elements need to be updated, the update_ui semaphore is given to trigger UI-related tasks.

FreeRTOS is chosen for its efficiency in managing multiple tasks in real-time embedded systems. It provides features like task scheduling, synchronization primitives (such as semaphores), and memory management tailored for resource-constrained environments. Semaphores, in particular, are essential for coordinating access to shared resources and signaling events between tasks without busy-waiting or excessive CPU usage.

In addition to the tasks described above, the program includes a reset task (reset_task) responsible for monitoring the state of a reset button (RESET_BUTTON). When the reset button is pressed, this task resets certain parameters of the system, such as the flicker interval, and signals a UI update to reflect the reset state on the LCD.

The use of semaphores in this program ensures proper synchronization and communication between tasks, enabling efficient task coordination without the need for busy-waiting or excessive CPU usage. Each task operates independently, responding to button events and updating the UI as necessary. This approach allows for modular and scalable code design, making it easier to manage and extend the functionality of the multitasking application.

1.2 Block Diagram

The following diagram shows the setup function of the program. Here we can see how each task is created using the function xCreateTask provided by the Arduino_FreeRTOS.

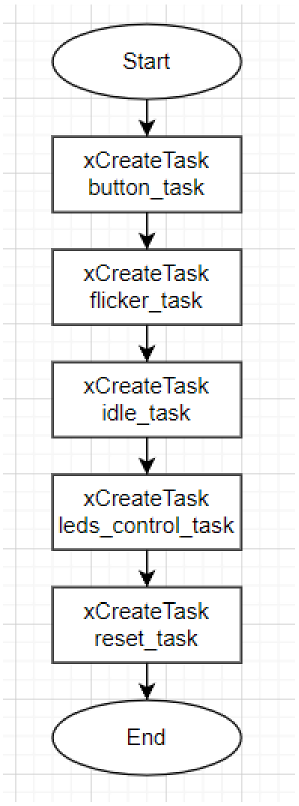


Figure 1. Block diagram of the program

Next, we have the diagram of the first task – button task. It is responsible for tracking the state of the led button. If it is pressed, the according functionality executes.

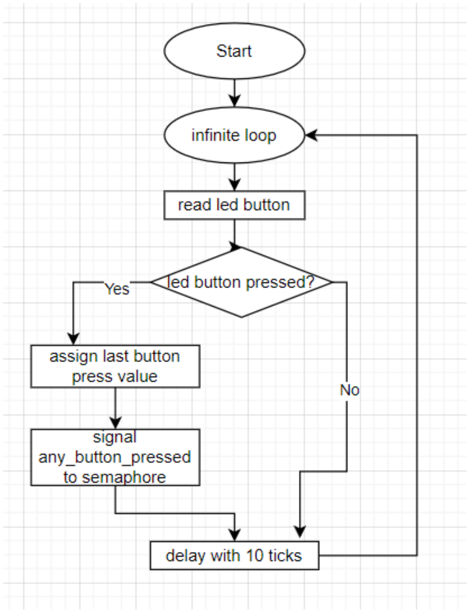
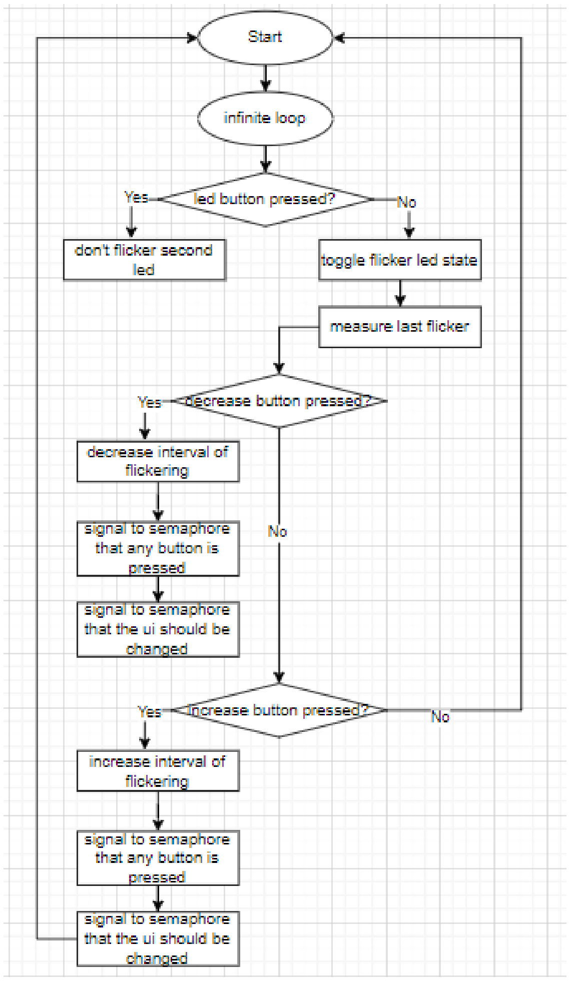


Figure 2. Block diagram of the button task

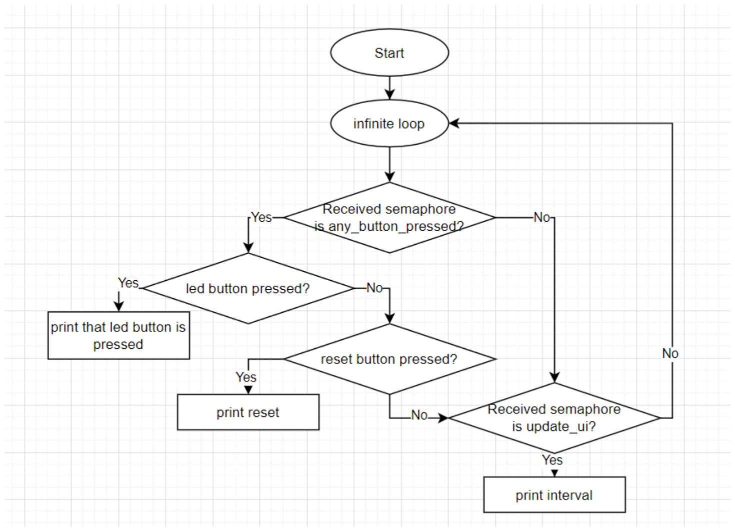
In the next figure there is presented the flow chart of the flicker task. It handled different functions: tracking the state of the increment/decrement buttons, tracking the led button, delegating semaphores with the xSemaphoreGive function.



Start
infinite loop
led button pressed?
Yes
No
don't flicker second led
toggle flicker led state
measure last flicker
decrease button pressed?
Yes
decrease interval of flickering
signal to semaphore that any button is pressed
signal to semaphore that the ui should be changed
No
increase button pressed?
Yes
No
increase interval of flickering
signal to semaphore that any button is pressed
signal to semaphore that the ui should be changed

Figure 3. Block diagram of flickering task

After that, there is the flow chart of the idle task. It is responsible for displaying on the LCD the functions that are executed by the system.



Start
infinite loop
Received semaphore is any_button_pressed?
Yes
No
led button pressed?
Yes
No
print that led button is pressed
reset button pressed?
Yes
No
print reset
Received semaphore is update_ui?
No
Yes
print interval

Figure 4. Block diagram of idle task

Lastly, there is the diagram of the additional task: reset task. This function resets the time of the flickering of the second led. The default value is 1000ms.

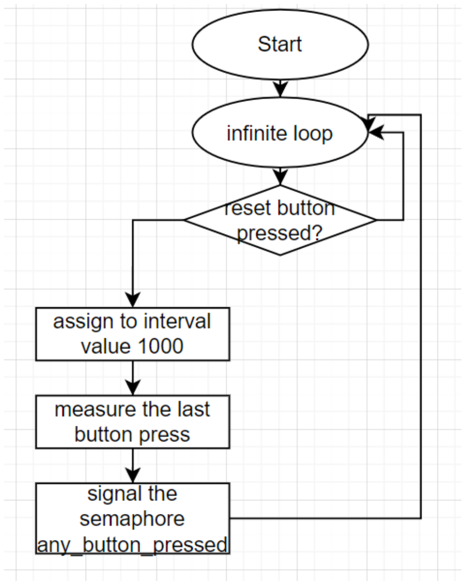

Start
infinite loop
reset button pressed?
assign to interval value 1000
measure the last button press
signal the semaphore any_button_pressed

Figure 5. Block diagram of reset task

1.3 Electrical Schematic

To simulate the electrical schematic for this project, I began by placing an Arduino Uno board at the center. I then connected two LEDs, one red and another one – yellow to 420 Ω resistors. Additionally, I added four push buttons, that were also connected to a 4.7kΩ resistors, with the other end of the resistors connected to the 5V pin of the Arduino. Ground connections for the buttons were made to the ground (GND) pin of the Arduino. Finally, I connected the cathode of each LED to the respective ground pin of the Arduino to complete the circuit. This schematic ensures proper interaction between the Arduino board, LEDs, and push buttons in the project.

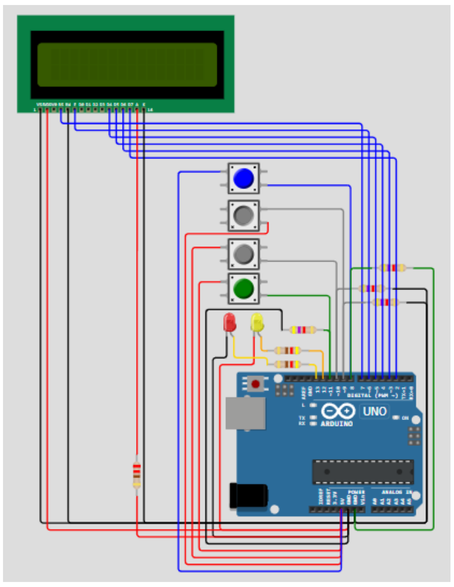


Figure 6. Electric circuit

1.4 Running Simulation

First of all, there is the simulation in its default state – when the red led is flickering. The LCD is started (nothing is shown yet), and the other led is turned off.

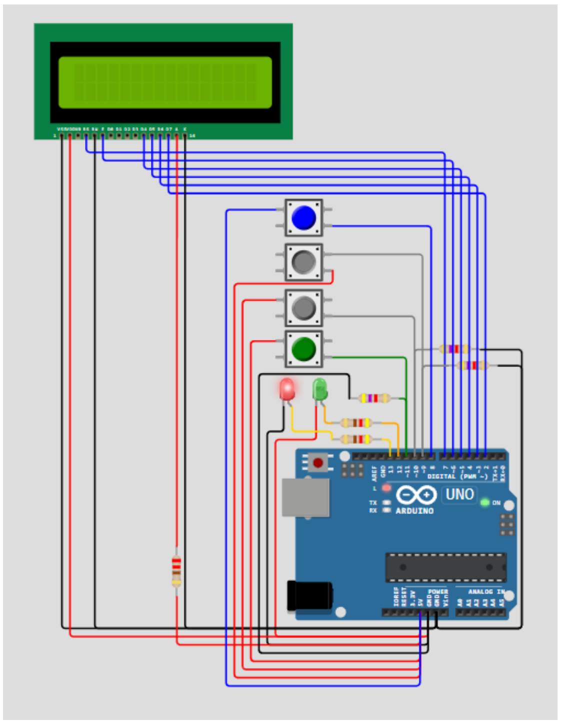


Figure 7. Simulation in the initial state

Here we can see the simulation in the state when the led button is pressed, and the second button (the yellow one), is turned on. In this state, the red button, which should blink, is now turned off.

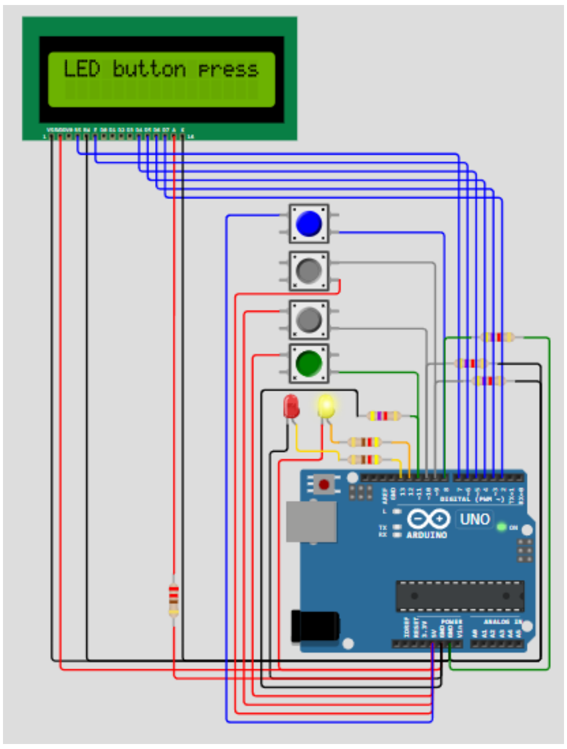


Figure 8. Simulation when led button is pressed

After that, I showed the simulation in the state when one of the buttons for increase/decrease flickering time is pressed. With every press on one of these buttons, the current changing interval of the flickering of the red led is displayed on the LCD (the number represent milliseconds).

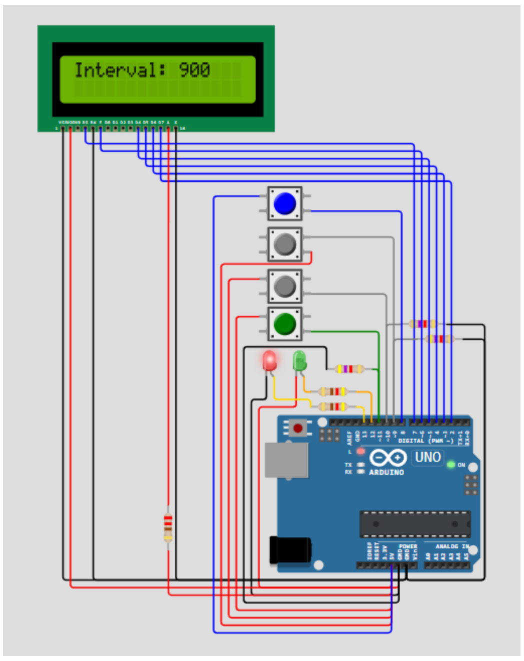


Figure 9. Simulation when increase button is pressed

And finally, there is the last task – reset task. Here we can see that when the fourth button of the circuit is pressed (the reset button), the interval of flickering becomes 1000ms and the corresponding message ("reset") is displayed on the LCD.

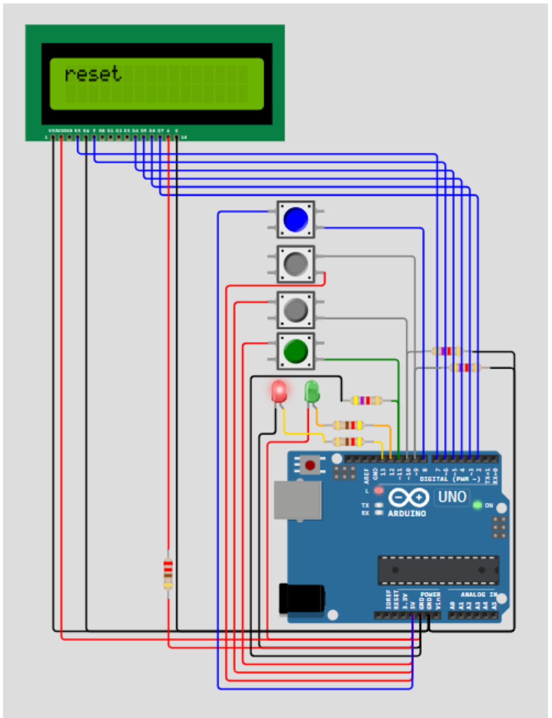


Figure 10. Simulation when reset button is pressed

CONCLUSION

For this lab assignment, I worked on developing an application that uses FreeRTOS to control at least three consecutive activities for a microcontroller unit (MCU). These jobs were created to handle a variety of functions, such as controlling a secondary LED that flashes when the primary LED is inactive, switching LED states based on button pushes, and modifying the recurrence time of the flashing LED through button inputs. Furthermore, I created an idle job that uses a provider/consumer framework to handle messages upon button pushes and maintain programme states, including LED status displays. My goal in integrating FreeRTOS was to give the MCU application a responsive and effective multitasking environment.

I was able to obtain hands-on experience with FreeRTOS real-time event processing, resource allocation, and task management during this project. I gained a deeper grasp of multitasking principles and how they are applied in embedded systems by coordinating many tasks. Furthermore, the Idle task's implementation for message management and status display demonstrated how flexible and adaptive FreeRTOS is at handling system interactions and states. All in all, this lab gave me invaluable practical experience creating real-time operating system (RTOS) apps and gave me the tools I needed to tackle future embedded system multitasking tasks.

APPENDIX A: Source code

```
#include <Arduino.h>
#include <Adafruit_LiquidCrystal.h>
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

const uint8_t LED_BUTTON = 11;
const uint8_t DECREASE_BUTTON = 10;
const uint8_t INCREASE_BUTTON = 9;
const uint8_t RED_LED = 13;
const uint8_t YELLOW_LED = 12;
const uint8_t rs = 7, en = 6, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
const uint8_t LCD_COLUMNS = 16;
const uint8_t LCD_ROWS = 2;
const uint8_t RESET_BUTTON = 8;
Adafruit_LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

bool button_pressed = false;
bool flicker_led_on = false;
bool decrease_button_pressed = false;
bool increase_button_pressed = false;
bool reset_button_pressed = false;
bool should_ticks_show = false;
uint32_t last_flicker = 0;
uint32_t interval = 1000;
uint32_t last_button_press = 0;

auto any_button_pressed = xSemaphoreCreateBinary();
auto update_ui = xSemaphoreCreateBinary();
auto reset_button_ui = xSemaphoreCreateBinary();

void init_lcd()
{
  lcd.begin(LCD_COLUMNS, LCD_ROWS);
  lcd.clear();
}

// Task to monitor the LED button
void button_task(void *pvParameters)
{
  for (;;)
  {
    // Read the state of the LED button
    button_pressed = digitalRead(LED_BUTTON);

    // If the button is pressed, signal the button press
    if (button_pressed)
    {
      last_button_press = millis();
      xSemaphoreGive(any_button_pressed); // Signal button press
      xSemaphoreGive(update_ui);          // Signal UI update
    }

    // Delay before checking the button state again
    vTaskDelay(pdMS_TO_TICKS(10));
  }
}

// Task to control LED flickering and handle button presses
void flicker_task(void *pvParameters)
{
  for (;;)
  {
    // Control LED flickering when the LED button is not pressed
    if (!button_pressed)
    {
      if (millis() - last_flicker > interval)
      {
        flicker_led_on = !flicker_led_on;
        last_flicker = millis();
      }
    }
    else
    {
      flicker_led_on = false;
    }

    // Check for decrease button press
    decrease_button_pressed = digitalRead(DECREASE_BUTTON);
    if (digitalRead(DECREASE_BUTTON))
    {
      interval = max(interval - 100, 200); // Adjust interval with a lower
       bound
      should_ticks_show = true;
      last_button_press = millis();
      xSemaphoreGive(any_button_pressed); // Signal button press
      xSemaphoreGive(update_ui);          // Signal UI update
      vTaskDelay(pdMS_TO_TICKS(200));     // Debouncing delay
    }

    // Check for increase button press
    increase_button_pressed = digitalRead(INCREASE_BUTTON);
    if (digitalRead(INCREASE_BUTTON))
    {
      interval += 100; // Adjust interval
      should_ticks_show = true;
      last_button_press = millis();
      xSemaphoreGive(any_button_pressed); // Signal button press
      xSemaphoreGive(update_ui);          // Signal UI update
      vTaskDelay(pdMS_TO_TICKS(200));     // Debouncing delay
    }

    // Delay before the next iteration
    vTaskDelay(pdMS_TO_TICKS(10));
  }
}

// Task to handle LCD display and user interface
void idle_task(void *pvParameters)
{
  vTaskDelay(pdMS_TO_TICKS(500));
  for (;;)
  {
    if (xSemaphoreTake(any_button_pressed, pdMS_TO_TICKS(200)))
    {
      if (button_pressed)
      {
        lcd.setCursor(0, 0);
        lcd.print("LED button pressed");
      }
      else if (reset_button_pressed)
      {
        lcd.setCursor(0, 0);
        lcd.print("reset");
        reset_button_pressed = false;
      }
    }
    if (xSemaphoreTake(update_ui, pdMS_TO_TICKS(200)))
    {
      if (should_ticks_show)
      {
        lcd.setCursor(0, 0);
        lcd.print("Interval: ");
        lcd.print(interval);
        should_ticks_show = false;
      }
    }
    vTaskDelay(pdMS_TO_TICKS(200));
    lcd.clear();
  }
}

// Task to control LED states
void leds_control_task(void *pvParameters)
{
  for (;;)
  {
    // Control the state of the LEDs
    digitalWrite(YELLOW_LED, button_pressed);
    digitalWrite(RED_LED, flicker_led_on);
    vTaskDelay(pdMS_TO_TICKS(10)); // Delay before the next iteration
  }
}

// Task to handle flicker reset button press and LCD update
void reset_task(void *pvParameters)
{
  for (;;)
  {
    // Check for reset button press
    reset_button_pressed = digitalRead(RESET_BUTTON);
    if (reset_button_pressed)
    {
      interval = 1000;
      last_button_press = millis();
      xSemaphoreGive(any_button_pressed); // Signal button press
      // xSemaphoreGive(update_ui);           // Signal UI update
    }
    // Delay before the next iteration
    vTaskDelay(pdMS_TO_TICKS(10));
  }
}

void setup()
{
  pinMode(YELLOW_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);

  init_lcd();

  xTaskCreate(button_task, "Button Task", 64, NULL, 1, NULL);
  xTaskCreate(flicker_task, "Flicker Task", 128, NULL, 1, NULL);
  xTaskCreate(idle_task, "UI Task", 256, NULL, 1, NULL);
  xTaskCreate(leds_control_task, "UI LEDs Task", 64, NULL, 1, NULL);
  xTaskCreate(reset_task, "Reset Task", 128, NULL, 1, NULL);
  vTaskStartScheduler();
}

void loop()
{
}
```

BIBLIOGRAPHY

[1] xTaskCreate, TASK CREATION, [Quoted: 10.03.2024], access link: https://www.freertos.org/a00125.html

[2] vTaskDelay, TASK CONTROL, [Quoted: 10.03.2024], access link: https://www.freertos.org/a00127.html

[3] Basic exemple of FreeRTOS with Arduino, void loop Robotech & Automation, [Quoted: 9.03.2024], access link https://www.youtube.com/watch?v=BuRGD3x-QDM&list=PLOYsAys6a6mmoyI2l440Wm5JwYhmtci8g&index=2

[4] FreeRTOS docs, [Quoted: 11.02.2024], access link: https://www.freertos.org/a00125.html

[5] Rtos multitasking tutorial, [Quoted: 01.01.2022], access link: https://www.youtube.com/watch?v=WQGAs9MwXno&ab_channel=SimplyExplained

[6] Manage FreeRTOS tasks - Suspend, Delay, Resume, Delete, [Quoted: 01.01.2022], access link: https://www.youtube.com/watch?v=jJaGRCgDo9s&ab_channel=SimplyExplained