Linnæus University
Sweden

Assignment 3

Real-time Hardware Project with RTOS

Embedded systems

Author: Daun, Michael
Examiner:Fredrik Ahlgren
Semester: 24VT
Course code: 1DT302

Linnæus University
Sweden

Contents

Introduction **3**

Hardware **4**

Functionality **5**

LEDs *5*

The Button *6*

The Sensor *6*

The Setup with Micropython *7*

Testing the Project **8**

Conclusion **9**

Sources **10**

Introduction

During this assignment, I encountered several software and coding-related challenges that significantly influenced the project. I want to highlight this early on, as some of my design choices were made either to avoid these difficulties or to work around the limitations they imposed.

I am a MacBook user with Apple's own silicon, the M1 processor. Since there are known issues with the ARM architecture of the M1 processor and certain applications (FreeRTOS being one of them), I was directed to use the Zephyr RTOS (Real-Time Operating System). Fortunately, there were no issues installing Zephyr on my hardware, and most dependencies were already installed since before.

Although compiling and coding did not pose significant problems, I faced major knowledge gaps and encountered completely new concepts, such as the usage of Device Trees. Consequently, there were some software functions and hardware components that I couldn't get to work, such as the UART (Universal Asynchronous Receiver/Transmitter). Previously, I had read print statements from the microcontroller using the terminal application Minicom. I couldn't get this to work, which made debugging harder. Furthermore, I couldn't get my basic 4-digit display or any buzzer to function. This is why one of my LED lamps is designated as a ‘debug_led’. I had to create a primitive system for debugging that flashed the LED lamps in a specific pattern depending on where in the code the error originated.

These limitations greatly affected my project. For example, if I wanted to measure time, I couldn't display it in my terminal or on my 4-digit display. Therefore, I decided to create a project where the results would be immediately visible during the live demonstration. I opted to have a few LED lamps flashing as a result of some kind of computation, with a button to control the tempo of the flashes and a flame sensor to act as an emergency stop when a flame was detected. By utilizing the more advanced thread support in the Zephyr OS, I was able to create a system that would react immediately, whereas the equivalent in the MicroPython implementation would suffer from a noticeable delay.

Hardware

A Raspberry Pi Pico was used as the microcontroller. The Pico provided its 3.3V and GND connections to the breadboard's power rails, which powered three LEDs, one button, and a sensor.

My sensor of choice was the KY-026 Flame Sensor. This sensor operates by detecting the wavelengths of light emitted by open flames [1]. The digital output is binary, sending a HIGH signal when detecting a flame. The analog output was not used in this project. The VCC and GND were connected to their respective power rails on the breadboard. The digital output from the sensor was connected to GPIO 17. The Pico's internal pull-down resistor was enabled for the sensor input. The sensor also includes a built-in potentiometer that controls its sensitivity. Since my usage mainly involved flashing a lighter above the sensor, I chose the lowest sensitivity setting.

The LEDs were connected to GPIO 1, 3, and 4, each in series with a 330 Ohm resistor. Finally, the button was connected to GPIO 16, with an external 10k Ohm pull-down resistor added to ensure that the voltage is pulled down during the button's idle state. The complete setup can be seen in Figure 1.

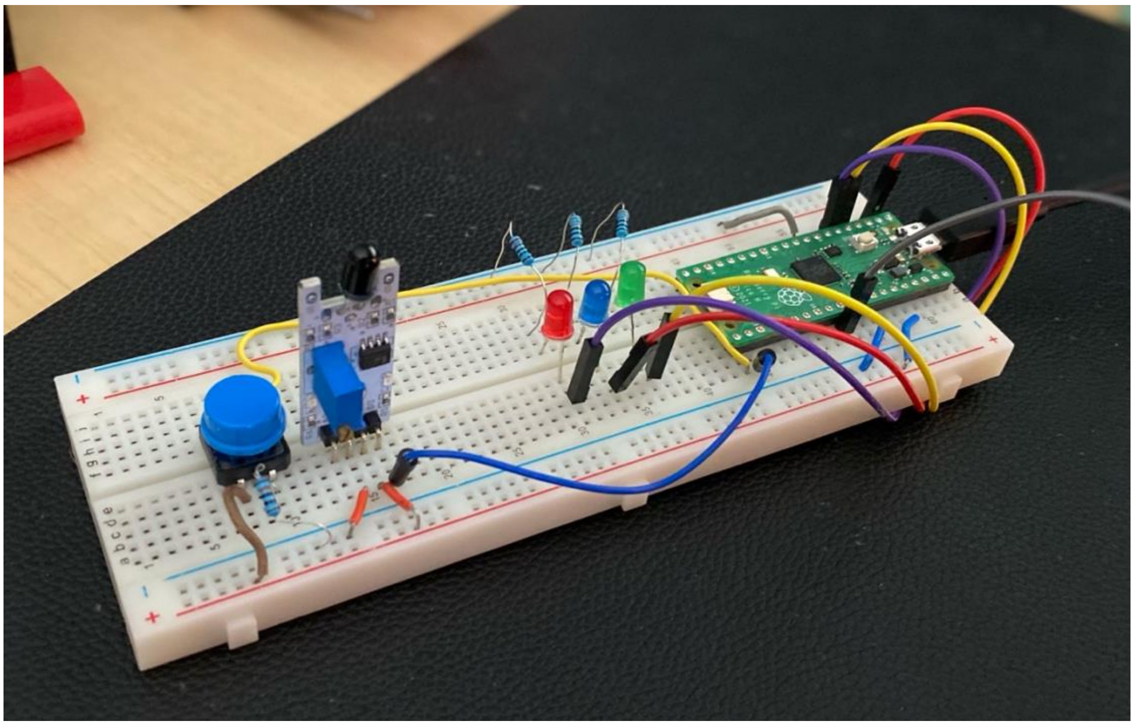


Figure 1. Breadbord with components for RTOS project.

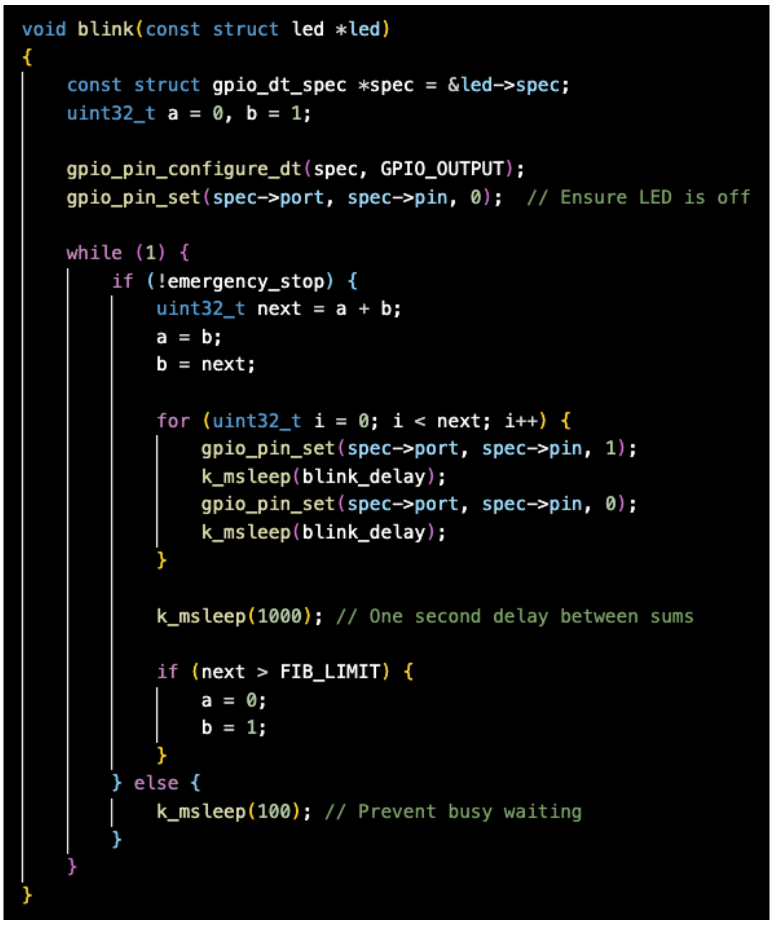
Functionality

LEDs

There are three LEDs that blinks at a given interval. Apart from a short initializing sequence when the device reboots, the amount of time each LED blinks is a result of the Pico calculating the sum of the classic Fibonacci sequence [2].

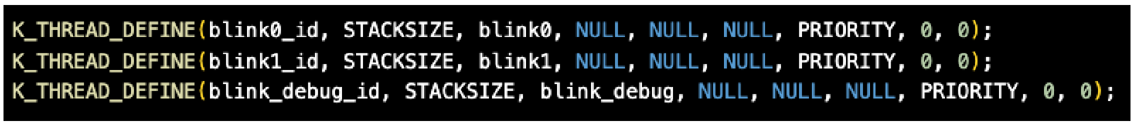
$$F(n) = F(n-1) + F(n-2)$$
$$With\ initial\ conditions:$$
$$F(0) = 0$$
$$F(1) = 1$$

Each LED has its own thread assigned to it, allowing for custom computations for each LED. As the author has a severe lack of imagination all the LEDs follow the same version of the Fibonacci sequence. The code for the function that the threads running the LEDs use is a modified version of the "blink" example from the Zephyr website.



```c
void blink(const struct led *led)
{
    const struct gpio_dt_spec *spec = &led->spec;
    uint32_t a = 0, b = 1;

    gpio_pin_configure_dt(spec, GPIO_OUTPUT);
    gpio_pin_set(spec->port, spec->pin, 0);  // Ensure LED is off

    while (1) {
        if (!emergency_stop) {
            uint32_t next = a + b;
            a = b;
            b = next;

            for (uint32_t i = 0; i < next; i++) {
                gpio_pin_set(spec->port, spec->pin, 1);
                k_msleep(blink_delay);
                gpio_pin_set(spec->port, spec->pin, 0);
                k_msleep(blink_delay);
            }

            k_msleep(1000); // One second delay between sums

            if (next > FIB_LIMIT) {
                a = 0;
                b = 1;
            }
        } else {
            k_msleep(100); // Prevent busy waiting
        }
    }
}
```

Figure 2. Code for the blink function in the Zephyr version.

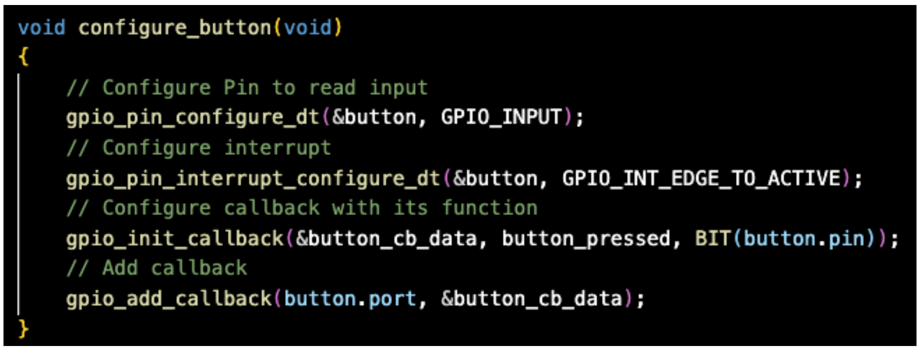


```c
K_THREAD_DEFINE(blink0_id, STACKSIZE, blink0, NULL, NULL, NULL, PRIORITY, 0, 0);
K_THREAD_DEFINE(blink1_id, STACKSIZE, blink1, NULL, NULL, NULL, PRIORITY, 0, 0);
K_THREAD_DEFINE(blink_debug_id, STACKSIZE, blink_debug, NULL, NULL, NULL, PRIORITY, 0, 0);
```

Figure 3. Threads associated with each LED. Priority is set to 7.

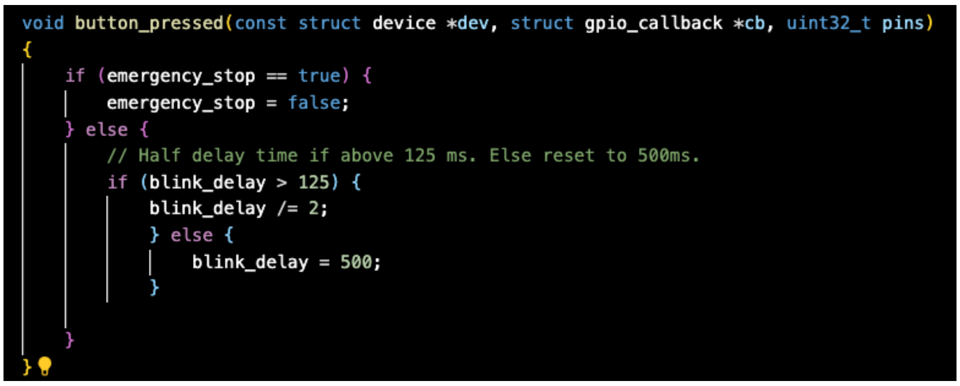
The Button

The button is configured with an interrupt that performs one of two actions: it either changes the delay between the LED blinks or resets the emergency status when the sensor has triggered an emergency. The code for the button interrupt could be improved with the addition of logic to eliminate contact bounces. However, contact bounces have not been an issue in this project.



```c
void configure_button(void)
{
    // Configure Pin to read input
    gpio_pin_configure_dt(&button, GPIO_INPUT);
    // Configure interrupt
    gpio_pin_interrupt_configure_dt(&button, GPIO_INT_EDGE_TO_ACTIVE);
    // Configure callback with its function
    gpio_init_callback(&button_cb_data, button_pressed, BIT(button.pin));
    // Add callback
    gpio_add_callback(button.port, &button_cb_data);
}
```

Figure 2. Code for configuration of the button interrupt.



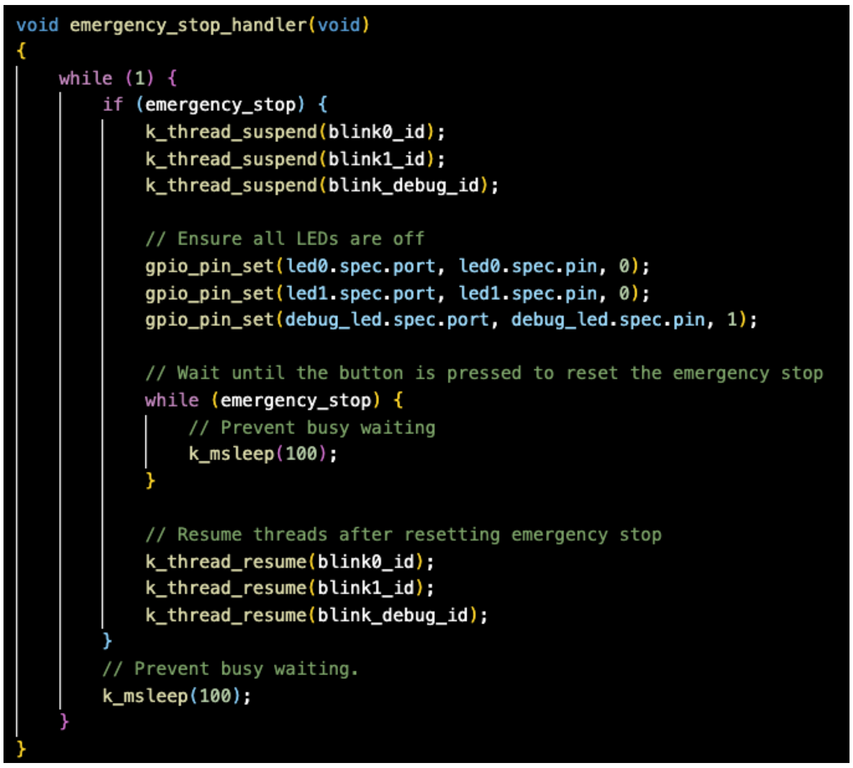
```c
void button_pressed(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    if (emergency_stop == true) {
        emergency_stop = false;
    } else {
        // Half delay time if above 125 ms. Else reset to 500ms.
        if (blink_delay > 125) {
            blink_delay /= 2;
        } else {
            blink_delay = 500;
        }

    }
}
```

Figure 3. Code for when button interrupt has been triggered.

The Sensor

The sensor is set up with an interrupt similar to the button. The logic that acts when an emergency is detected, like the LEDs, has its own thread assigned. However, the sensor's thread has a priority of 1, whereas the LEDs have their priority set to 7. The sensor is intended to act as an emergency stop, and thus, if a flame is detected, this thread should have the highest priority.

With this sensor, where all logic is already contained in the circuit it's soldered on, all we need to extract from it is a binary value, either HIGH or LOW. In the device tree, the sensor is mapped as a button due to its simplicity. The code for the thread that handles the emergency checks if the sensor has been triggered. If so, the thread will suspend all the LED threads immediately and force the debug LED to stay lit to signal that an emergency has been triggered. The threads will resume once the emergency is resolved, i.e., when the button has been pressed.



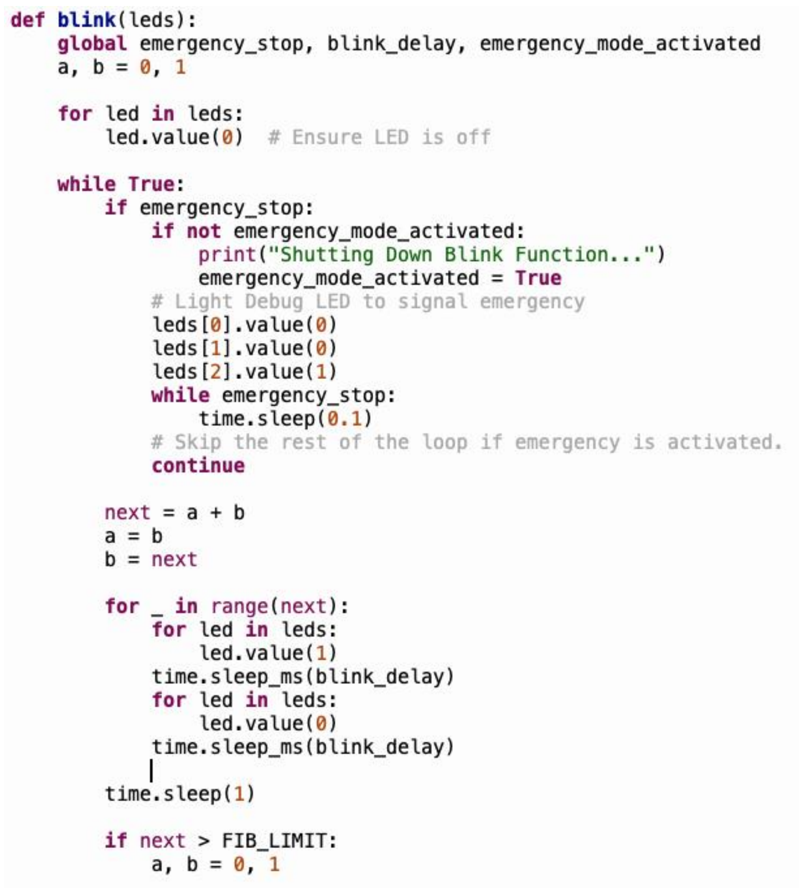
```c
void emergency_stop_handler(void)
{
    while (1) {
        if (emergency_stop) {
            k_thread_suspend(blink0_id);
            k_thread_suspend(blink1_id);
            k_thread_suspend(blink_debug_id);

            // Ensure all LEDs are off
            gpio_pin_set(led0.spec.port, led0.spec.pin, 0);
            gpio_pin_set(led1.spec.port, led1.spec.pin, 0);
            gpio_pin_set(debug_led.spec.port, debug_led.spec.pin, 1);

            // Wait until the button is pressed to reset the emergency stop
            while (emergency_stop) {
                // Prevent busy waiting
                k_msleep(100);
            }

            // Resume threads after resetting emergency stop
            k_thread_resume(blink0_id);
            k_thread_resume(blink1_id);
            k_thread_resume(blink_debug_id);
        }
        // Prevent busy waiting.
        k_msleep(100);
    }
}
```

Figure 4. Logic for the thread responsible for the handling the emergency stop.

The Setup with Micropython

Without an operating system, we can’t use the same approach as with the Zephyr version. We are limited to two threads, one per core on the Raspberry Pi Pico. In the Zephyr version, each LED could potentially have custom behaviors due to the ability to assign individual threads. However, in the non-RTOS version with MicroPython, we are significantly more restricted. If we implemented custom computations for each LED in the Zephyr version, replicating this functionality in the MicroPython setup would not be feasible.

Nevertheless, the sensor and the button can still be set up as interrupts with the same functions as before. The main difference is that we can’t suspend any threads like we did with Zephyr. This means the threads need to be polling, constantly checking if an emergency has been triggered. By this logic, the thread responsible for blinking the LEDs can only check if the emergency has been triggered after it has finished blinking its current iteration of the Fibonacci sequence.



```python
def blink(leds):
    global emergency_stop, blink_delay, emergency_mode_activated
    a, b = 0, 1

    for led in leds:
        led.value(0)  # Ensure LED is off

    while True:
        if emergency_stop:
            if not emergency_mode_activated:
                print("Shutting Down Blink Function...")
                emergency_mode_activated = True
            # Light Debug LED to signal emergency
            leds[0].value(0)
            leds[1].value(0)
            leds[2].value(1)
            while emergency_stop:
                time.sleep(0.1)
            # Skip the rest of the loop if emergency is activated.
            continue

        next = a + b
        a = b
        b = next

        for _ in range(next):
            for led in leds:
                led.value(1)
            time.sleep_ms(blink_delay)
            for led in leds:
                led.value(0)
            time.sleep_ms(blink_delay)

        time.sleep(1)

        if next > FIB_LIMIT:
            a, b = 0, 1
```

Figure 5. Logic for LEDs with non-RTOS version.

Testing the Project

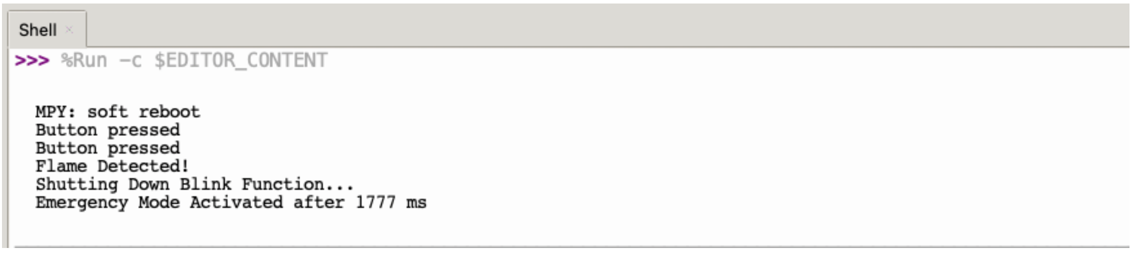
As mentioned in the introduction, the inability to receive communication through the UART means that the microcontroller can't provide any time units to show the actual reaction time for the Zephyr implementation. Therefore, testing has been conducted manually, with the goal of ensuring the reaction appears "instantaneous."

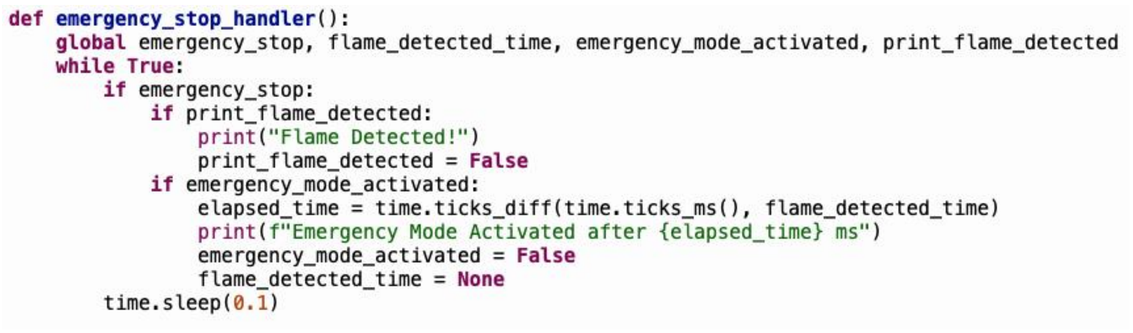
The testing consisted of the following steps:

1. Verify that the microcontroller initializes with the sequence of flashing LEDs in its start-up pattern.
2. Ensure the microcontroller flashes LEDs as specified in the "blink" function.
3. Confirm that the button can control the delay between flashes for the LEDs.

4. Check that the emergency sensor reacts to a lit flame or spark in proximity to the sensor. This is confirmed when all blinking stops, and the debug LED lights up.

Although the Zephyr version couldn't measure the reaction time, the non-RTOS version can provide feedback to the terminal. The testing process remained the same, with the addition of evaluating the terminal output from the non-RTOS implementation. The measured reaction time was never instantaneous as with the Zephyr version. It depended on the current Fibonacci number the LEDs were set to output and the delay between flashes set by the button. I recorded the reaction a few times, with values ranging from 1 second to 6 seconds.


```
Shell
>>> %Run -c $EDITOR_CONTENT

  MPY: soft reboot
  Button pressed
  Button pressed
  Flame Detected!
  Shutting Down Blink Function...
  Emergency Mode Activated after 1777 ms
```
Figure 6. Terminal feedback from the non-RTOS version.


```python
def emergency_stop_handler():
    global emergency_stop, flame_detected_time, emergency_mode_activated, print_flame_detected
    while True:
        if emergency_stop:
            if print_flame_detected:
                print("Flame Detected!")
                print_flame_detected = False
            if emergency_mode_activated:
                elapsed_time = time.ticks_diff(time.ticks_ms(), flame_detected_time)
                print(f"Emergency Mode Activated after {elapsed_time} ms")
                emergency_mode_activated = False
                flame_detected_time = None
        time.sleep(0.1)
```
Figure 7. Function that measures reaction time for the non-RTOS version.

Conclusion

The purpose of this assignment was to create a microcontroller-based project with real-time constraints that necessitated the use of an RTOS. Early on, I decided to develop an emergency system. The use of the flame sensor is potentially lifesaving and crucial for many systems we use daily. The flame sensor has the capability to react to a spark alone, which could prevent catastrophic events in many systems, such as an airplane.

I believe that the usage demonstrated in this assignment highlights the necessity of using an RTOS for time-critical operations. The somewhat trivial computation performed by the LEDs could be replaced with any other task. Regardless of the specific tasks, without an operating system to manage and control which process is doing what, time-critical systems will inevitably suffer.

Sources

[1] "KY-026 Flame Sensor Module," Electrothinks, Jan. 2021. [Online]. Available: https://www.electrothinks.com/2021/01/KY-026-flame-sensor-module.html. [Accessed: 24-May-2024].
[2] "Fibonacci sequence," Wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Fibonacci_sequence. [Accessed: 24-May-2024].