# Zephyr RTOS Project Report

107474-Joseane Pereira
109050-Gabriel Costa
Universidade de Aveiro, DETI

December 26, 2024

# Contents

# 1 Introduction

The aim of this project is to apply the Linux Real-Time Services and the Real-Time Model to the development of a real-life inspired real-time application. The project encompasses a set of cooperating tasks, involving synchronization, shared resources, access to a real-time database, etc.

# 2 Architecture

## 2.1 System Overview

In this project, the real-time monitoring system is structured around several primary tasks. These tasks were scheduled using a Static Table-Based Scheduler, which is the most important module of this project.

Given our use-case, we also implemented a RTDB, in order to have a structure that could store the changes in information.

## 2.2 Task

We designed a structure named **Task** to encapsulate all the necessary information about a task. This includes attributes such as its period (measured in ticks), execution time, and other relevant details.

This information is then used in the STBS to store and manage the information of each task of the system.

## 2.3 Static Table-Based Scheduler Implementation

### 2.3.1 Initialization

To initialize our STBS, we define the time each micro-cycle has, the maximum number of tasks, initialize the number of tasks that the table currently has, and finally, initialize a list of the "Tasks" of the system.

### 2.3.2 STBS_AddTask

This function is only responsible for initializing a new "Task" and adding it to the Table, making sure it doesn't exceed the maximum number of tasks.

### 2.3.3 STBS_Start

The function **STBS_Start** is invoked after all tasks have been added to the scheduler. Its primary responsibility is to determine whether the tasks are schedulable and, if so, to handle their scheduling and execution. Specifically, this involves assigning each task to its respective micro-cycle.

Once the scheduling is complete, the function iterates through the micro-cycles, resuming and executing all tasks within each cycle in the predefined order. This process continues until the program is stopped.

#### 2.3.4 STBS_print_content

This function is responsible for displaying the contents of the table generated by **STBS_Start** in a structured, table-like format. This presentation makes it easier to understand how the tasks are organized.

#### 2.3.5 STBS_destroy

This function is responsible for releasing the memory allocated for the STBS and the table it generated.

### 2.4 Real-Time Database (RTDB)

The RTDB is used to store the states of LEDs and buttons, ensuring synchronized access and updates by different tasks.

## 3 Tasks

### 3.1 Task Characteristics

The system is composed by the following tasks:

1. **Task 0**: Updates the RTDB with the button states.
2. **Task 1**: Updates the LED states based on the button states from the RTDB.
3. **Task 2**: Validates RTDB entries and resets them if they are corrupted.

Each task in the system is assigned a specific priority and a period to ensure timely execution and prevent task starvation. The following table outlines the task priorities and activation periods.

| Task | Priority | Activation Period | Description |
|---|---|---|---|
| Task 0 | 1 | 50ms | Updates the RTDB with the button states. |
| Task 1 | 2 | 100ms | Updates the LED states based on the button states from the RTDB. |
| Task 2 | 1 | 100ms | Validates RTDB entries and resets them if they are corrupted. |

Table 1: Task Priorities and Activation Periods

### 3.2 Execution Patterns and Relevant Events

The RTDB is critical for data synchronization:

- **Task 0** writes to the RTDB, while **Task 1** and **Task 2** read from it.
- Since there aren't multiple tasks trying to write to the RTDB there is no need to implement mutexes or other mechanisms to handle concurrent access.

# 4 Real-Time System Characterization

## 4.1 System Performance

## 4.2 System Schedulability

To confirm that all tasks meet their deadlines, we perform a schedulability analysis using the system's utilization factor.

**Utilization Factor (U)**: Given that each task is independent and periodic, we can calculate the utilization factor $U$ using:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

where $C_i$ is the computation time and $T_i$ is the period of each task. Assuming each task completes within its assigned period, this calculation helps ensure that the system remains schedulable.

For the tasks in our system, the utilization factor is calculated as follows:

$$U = \frac{20}{50} + \frac{25}{100} + \frac{30}{100} = 0.95$$

# 5 Use-Case Implementation

## 5.1 Smart I/O Module

Here, we implemented a mapping from each button to each led, meaning that each button will toggle the state of the respective led.

## 5.2 UART Communication Protocol

In order for the computer to communicate with the Microcontroller, we implemented a simple protocol. It consists in writing commands in the PC, and receiving an ACK message, confirming if the command was executed successfully or if something went wrong (e.g: command doesn't exist or wrong checksum). In our implementation, every time a command is completed (the user typed the character "#") the command is processed and executed (if prompted correctly), and the respective acknowledge is sent back as a response.

For this particular use-case,and project, we decided to process the command inside the UART interrupt function meaning that we assume that the overhead is close to 0, however, we understand that this is not the best practice, since it can add overhead to the execution of the other tasks.

# 6 Tests

The tests for the scheduler verify that tasks are added correctly and meet their deadlines. The test suite includes the following.

- **Scheduler Test**: Verifies that the scheduler can schedule a schedulable system and detects when the system is not schedulable.

- **Task Addition Test**: Verifies that tasks can be added and appear in the task table.
- **Task Deadline Test**: Verifies that tasks meet their deadlines.

## 6.1 Scheduler Test

When testing the scheduler, we noticed that it wasn't detecting when the tasks exceeded their deadline, so we had to make some small changes (which are marked in the code with "CHANGED"), namely, adding this condition

which checks if a task is delayed past it's deadline.

Now, for the tests, when creating the table, we tested if the tasks that executed in the same micro-cycle with the same priority, are ordered by their period, meaning that the ones with lower period execute first (EDF).

Then, we tested whether tasks with higher priority execute first.

Last but not least, we needed to test if the our code could recognize if a system is not schedulable. For that we did the following tests:

- We created a table with a micro-cycle of 50 ms and 4 tasks. These tasks had a period of 1, 3, 2, 2 ticks, priorities of 10, 5, 7, 2 and 20, 25, 30, 15 ms of execution time, respectively. Since the first task has a period of 1 tick and had the lowest priority, it was not able to meet its first deadline, which makes this system not schedulable.
- For the next test, we changed the priority of the first task to 1. This system is also not schedulable since the third task is not able to execute before it meets it's deadline, since in the first tick it executes the first and fourth task, and in the second one it executes the first and the second, leaving no time for the third task to execute

# 7 Results

The tests confirm that the scheduler is initialized correctly, tasks are added as expected, and all tasks meet their deadlines within the specified margins.

# 8 Conclusion

In this project, a multi-threaded system was successfully implemented to manage tasks in a real-time environment. The scheduler ensures that tasks are executed periodically and meet their deadlines. The RTDB provides synchronized access to shared data, preventing data corruption. Overall, the project demonstrates effective use of periodic task scheduling, synchronization, and real-time data management.