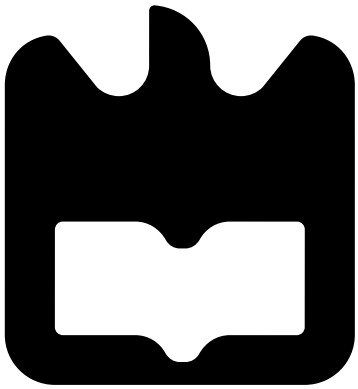


Zephyr RTOS Project Report

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

January 3, 2025

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 to 0, 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 STBS list of tasks, making sure it doesn't exceed the maximum number of tasks.

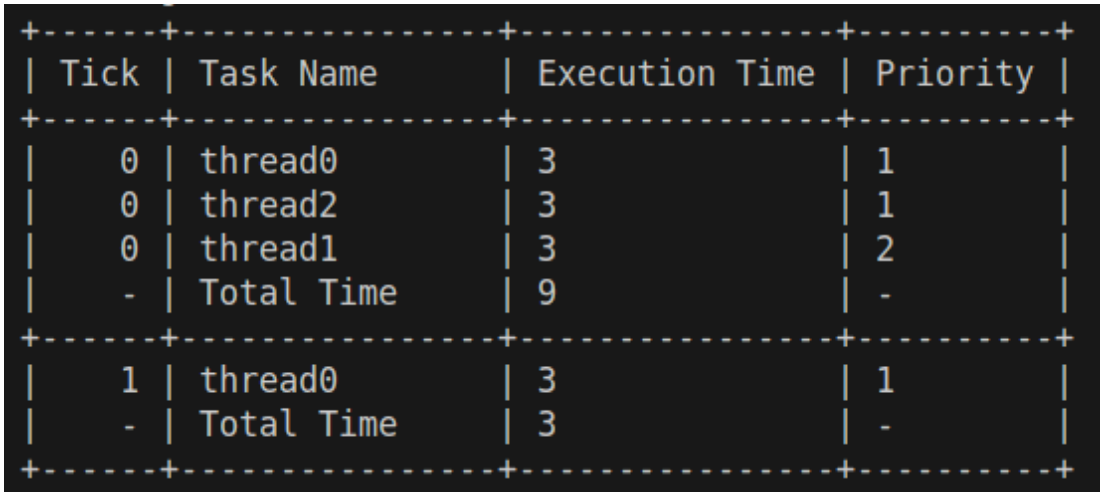
2.3.3 STBS_Start

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 scheduler table in a structured, table-like format.



Tick	Task Name	Execution Time	Priority
0	thread0	3	1
0	thread2	3	1
0	thread1	3	2
-	Total Time	9	-
1	thread0	3	1
-	Total Time	3	-

Figure 1: Scheduler's table

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.

3.2 Execution Patterns and Relevant Events

The RTDB is critical for data synchronization:

- **Task 0** is responsible for updating the RTDB with the current states of the buttons. It reads the states of the buttons and writes these states to the RTDB. This task runs periodically every 50ms.
- **Task 1** is responsible for updating the LED states based on the button states stored in the RTDB. It reads the button states from the RTDB and updates the LED states accordingly. This task runs periodically every 100ms.
- **Task 2** is responsible for validating the RTDB entries and resetting them if they are corrupted. It reads the states of the LEDs and buttons from the RTDB and checks for any invalid values. If any invalid values are found, it resets them to a default state. This task runs periodically every 100ms.
- 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.

The tasks communicate through the RTDB, which acts as a shared data structure. Task 0 writes the button states to the RTDB, and Tasks 1 and 2 read these states. Task 1 updates the LED states based on the button states, and Task 2 validates the RTDB entries. This communication pattern ensures that the tasks operate on the most recent data without conflicts.

4 Real-Time System Characterization

4.1 System Performance

To evaluate the system's performance, we analyzed both the task execution times and the scheduler's overhead.

Each task takes approximately 2 to 3 milliseconds to execute, depending on its complexity. The scheduler, when managing three tasks in a single micro-cycle, requires 8 to 9 milliseconds to complete. For cycles with only one task, the scheduler takes around 2 to 3 milliseconds. This demonstrates that the scheduler overhead is minimal, contributing only a few microseconds per task execution. Such low overhead indicates a highly efficient scheduler.

The measurements were obtained through the following modifications:

- In each task, the execution start and finish times were recorded using **k_uptime_get**. The difference between these timestamps was calculated and printed to determine the execution time of the task .
- In the scheduler, the start and finish times of each micro-cycle were recorded using the same function. The difference between these timestamps was printed to assess the scheduler's execution time.

These results confirm that the scheduler introduces negligible performance overhead, maintaining system responsiveness and ensuring real-time task scheduling efficiency.

4.2 System Schedulability

Since our system uses a Static Table-Based Scheduler (STBS), we added code to verify if the task set is schedulable. This code checks whether any task is delayed beyond its deadline. Also, the only shared resource is the Real-Time Database (RTDB), and because no more than one task modifies its values at the same time, there is no need to account for resource conflicts.

Our system operates with a 50ms tick. Based on the resulting schedule table 1, we can confirm that the task set is indeed schedulable..

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 of 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).

For this 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 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.
- **Scheduler organization**: Verifies that tasks are ordered by their priority and by their period.

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

```
if(tick+1 > stbs.macro_cycle
 || stbs.task_table[task_idx].delay_count >= stbs.task_table[task_idx].ticks){ // CHANGED
    printk("System not schedulable\n");
    for (int i = 0; i < stbs.macro_cycle; i++) {
        k_free(entry[i].tasks);
    }
    k_free(entry);
    return;
}
```

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.

Tick	Task Name	Execution Time	Priority
0	thread0	10	1
0	thread1	40	1
-	Total Time	50	-
1	thread0	10	1
1	thread3	10	1
1	thread2	30	1
-	Total Time	50	-
2	thread0	10	1
2	thread1	40	1
-	Total Time	50	-
3	thread0	10	1
3	thread3	10	1
3	thread2	30	1
-	Total Time	50	-
4	thread0	10	1
4	thread1	40	1
-	Total Time	50	-
5	thread0	10	1
5	thread3	10	1
-	Total Time	20	-

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

Tick	Task Name	Execution Time	Priority
0	thread3	7	2
0	thread2	12	5
0	thread1	15	7
0	thread0	10	10
-	Total Time	44	-
1	thread0	10	10
-	Total Time	10	-
2	thread3	7	2
2	thread1	15	7
2	thread0	10	10
-	Total Time	32	-
3	thread2	12	5
3	thread0	10	10
-	Total Time	22	-
4	thread3	7	2
4	thread1	15	7
4	thread0	10	10
-	Total Time	32	-
5	thread0	10	10
-	Total Time	10	-

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 results of our tests confirm that the scheduler is functioning as intended. Tasks are correctly initialized, added, and executed. Furthermore, all tasks meet their deadlines within the specified margins when the system is schedulable. Non-schedulable systems are accurately identified, validating the robustness of our scheduler.

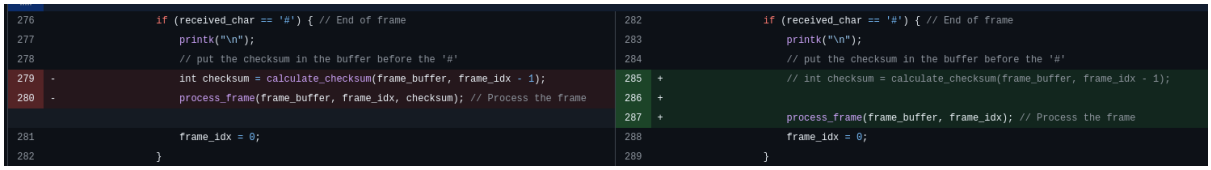
8 Conclusion

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.

9 Appendice

We made a change in the code regarding the use of checksums. Before, we didn't send the checksum correctly to the microcontroller. We were calculating the checksum when we had a complete frame and we would send it to the function who executes the command. To fix this, we made the following changes:



```
276                 if (received_char == '#') { // End of frame
277                     printk("\n");
278                     // put the checksum in the buffer before the '#'
279 -                   int checksum = calculate_checksum(frame_buffer, frame_idx - 1);
280 -                   process_frame(frame_buffer, frame_idx, checksum); // Process the frame
281                     frame_idx = 0;
282                 }
```

```
282                 if (received_char == '#') { // End of frame
283                     printk("\n");
284                     // put the checksum in the buffer before the '#'
285 +                   // int checksum = calculate_checksum(frame_buffer, frame_idx - 1);
286 +
287 +                   process_frame(frame_buffer, frame_idx); // Process the frame
288                     frame_idx = 0;
289                 }
```

Figure 2: Receiving checksum as an input instead of calculating it

```
83          // Calculate and validate checksum!
84  -       int received_checksum = checksum;
85  -       //printk("Received checksum: %d\n", received_checksum);
86  -       if (calculate_checksum(frame, frame_length - 1) != received_checksum) {
87  -           printk("Received checksum: %d  Calculated checksum: %d\n",
    received_checksum,calculate_checksum(frame, frame_length - 1));
88              send_ack('3'); // Checksum error
89              return;
90          }
```

```
89  +       const char *checksum = &frame[frame_length - 4];
90  +
91  +
92  +       int received_checksum = atoi(checksum);
93          // Calculate and validate checksum!
94  +
95  +       if (calculate_checksum(frame, frame_length-4) != received_checksum) {
96  +           printk("Received checksum: %d  Calculated checksum: %d\n",
    received_checksum,calculate_checksum(frame, frame_length - 4));
97              send_ack('3'); // Checksum error
98              return;
99          }
```

Figure 3: Compare the checksum calculated to the one received as an input