Tanguy **DEREN**
Laure **INTURRISI**
**4TS1**

# **FINAL ASSIGNMENT:** DESIGNING AN RTOS

Real-time Embedded Software

In422 | Mr. SINGH
**April 26th, 2024**

# Table of contents

## Abstract

This project focuses on the implementation of a Real-Time Operating System (RTOS) using FreeRTOS for embedded systems and mission-critical applications. The objective is to execute predefined tasks with minimal delay and fixed priority order, ensuring constant response times. Methods involve program initialization, task creation, and scheduling using FreeRTOS functions. Each task, ranging from simple actions like printing to complex computations, is designed to meet specific requirements. To evaluate system performance, the Worst Case Execution Time (WCET) is determined through repetitive task execution, with a safety margin added for robustness. The project demonstrates the practical application of RTOS principles in real-world scenarios.

## I. Introduction

To understand the importance of this project, we first need to understand what an RTOS is. It can be defined as follows: a Real-Time Operating System is an operating system designed to meet the stringent real-time requirements of embedded systems and mission-critical applications. Unlike traditional operating systems, which are generally designed to provide general task processing services with variable priority, an RTOS is optimised to guarantee predetermined and constant response times for real-time tasks.

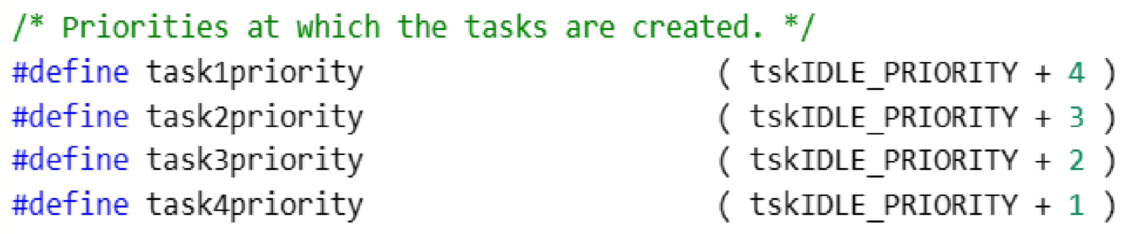
In our case, the challenge will be to carry out predefined tasks with a minimum delay and a fixed order of priority. The operation will then have to be repeated several times and we will have to decide on the most optimal WCET (Worst Case Execution Time) possible.

## II. Methods

To do this, we're going to use FreeRTOS. FreeRTOS is a very popular open-source real-time operating system designed specifically for embedded systems. It is lightweight, portable and provides the basic functionality needed to develop real-time applications, which more than meets our expectations.

## 1. Program initialization

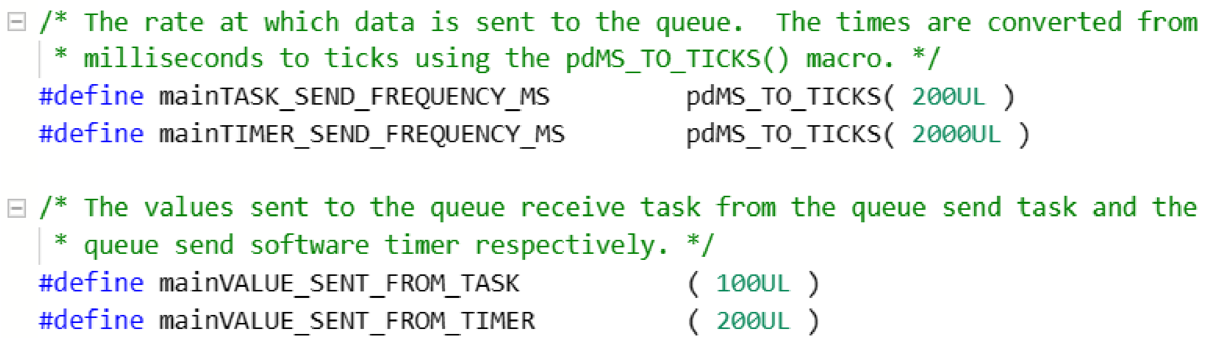
We start by defining the order of priority for each of our tasks in the order set out earlier. This is our code:



```
/* Priorities at which the tasks are created. */
#define task1priority                     ( tskIDLE_PRIORITY + 4 )
#define task2priority                     ( tskIDLE_PRIORITY + 3 )
#define task3priority                     ( tskIDLE_PRIORITY + 2 )
#define task4priority                     ( tskIDLE_PRIORITY + 1 )
```

The program will then execute the task first and so on until the program is restarted.

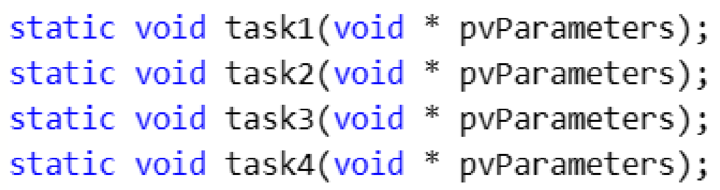
Next, we define the rate at which data is sent to and received from the queue. This represents the delay between successive executions of the task. The target data are the various tasks and the program's internal timer. We also change the milliseconds to ticks because there are several advantages. Firstly, the precision is greater for this type of real-time application. Also, ticks are often used as a unit of time in RTOSs because they are independent of the hardware clock frequency. This means that elapsed time is measured in terms of ticks rather than physical units of time, as in this case milliseconds. This makes the code portable between different hardware platforms without requiring any modifications.



```
/* The rate at which data is sent to the queue.  The times are converted from
 * milliseconds to ticks using the pdMS_TO_TICKS() macro. */
#define mainTASK_SEND_FREQUENCY_MS         pdMS_TO_TICKS( 200UL )
#define mainTIMER_SEND_FREQUENCY_MS        pdMS_TO_TICKS( 2000UL )

/* The values sent to the queue receive task from the queue send task and the
 * queue send software timer respectively. */
#define mainVALUE_SENT_FROM_TASK           ( 100UL )
#define mainVALUE_SENT_FROM_TIMER          ( 200UL )
```

We now move on to creating our tasks. To begin, we define the type of task functions, like so:



```
static void task1(void * pvParameters);
static void task2(void * pvParameters);
static void task3(void * pvParameters);
static void task4(void * pvParameters);
```

Then we can start programming our main function, which will store the various tasks. Inside, we start by creating the tasks with the following function, which we will describe in detail:

```
xTaskCreate( task1, "Rx", configMINIMAL_STACK_SIZE, NULL, task1priority, NULL );
xTaskCreate( task2, "TX", configMINIMAL_STACK_SIZE, NULL, task2priority, NULL );
xTaskCreate( task3, "FX", configMINIMAL_STACK_SIZE, NULL, task3priority, NULL );
xTaskCreate( task4, "ZX", configMINIMAL_STACK_SIZE, NULL, task4priority, NULL );
```

**`xTaskCreate()`** is a function that defines the characteristics of a task and adds it to the task scheduler so that it can be executed. "task" is the name of the function that defines the behavior of the task, "Rx" is the name of the task, which must be unique, and "task1priority" is the order of priority of the task defined at the beginning. The other characteristics are not so important to us and are more complex to understand.

We can now start scheduling the tasks. To do this, we use the following function:

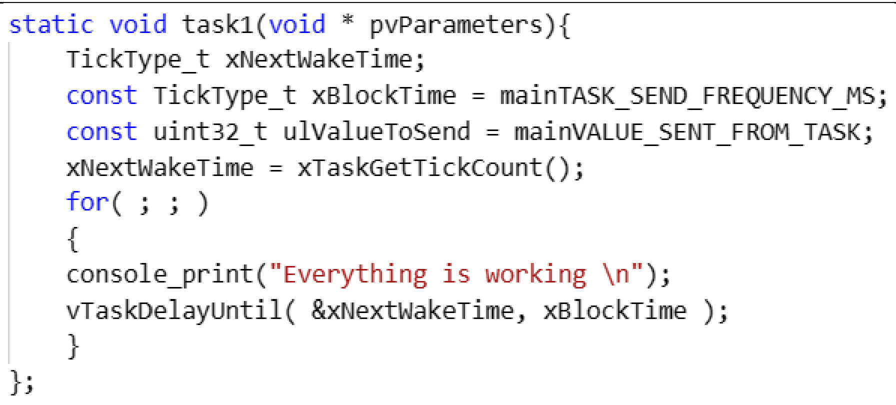
```
vTaskStartScheduler();
```

This function is used to start the FreeRTOS task scheduler, which is responsible for managing real-time tasks in the system. Before executing this function, it must be ensured that all the necessary system initializations have been carried out, such as task creation, task priority, etc. The function then starts the operating system and begins executing the tasks previously created. FreeRTOS then manages the tasks in real time according to their priority and order. It ensures that the tasks are carried out reliably and on time. Finally, once the function has been called, control of the execution flow is transferred to the FreeRTOS task scheduler. From this point onwards, the system operates in multitasking mode, where several tasks can run simultaneously according to their order of priority.

To finish this main function, we set up an infinite loop (which executes indefinitely until an action stops the execution of the program). This type of loop is often used in embedded systems or programs that require continuous execution, such as ours.

## 2. Creating task functions

To understand how tasks are created, let's look at task 1 in detail. To start with, here's task 1. We had to print "Working" or some other string that said everything was working normally.



```
static void task1(void * pvParameters){
    TickType_t xNextWakeTime;
    const TickType_t xBlockTime = mainTASK_SEND_FREQUENCY_MS;
    const uint32_t ulValueToSend = mainVALUE_SENT_FROM_TASK;
    xNextWakeTime = xTaskGetTickCount();
    for( ; ; )
    {
    console_print("Everything is working \n");
    vTaskDelayUntil( &xNextWakeTime, xBlockTime );
    }
};
```

We start by defining three local variables "`xNextWakeTime`", "`xBlockTime`" and "`ulValueToSend`". The first is used to store the time at which the next execution of the task should take place. The second represents the delay between successive executions of the task. Finally, the third is not used here, but can be a value to be sent to another task. To initialize "`xNextWakeTime`" we use the `xTaskGetTickCount()` function to allow it to store the number of ticks elapsed since the system was started. The variable will then be used to schedule future executions of the task.

Next, we initialize an infinite loop so that the task can perform its action. *Everything is working* is displayed, then the `vTaskDelayUntil()` function is called to put the task on hold. This waiting time is defined by "`xBlockTime`" from the time defined by "`xNextWakeTime`". In this way, future executions of the task can be scheduled with a constant delay.

The other three tasks work in the same way, apart from their actions of course. Task 2 converts a temperature from Fahrenheit to Celsius. Task 3 multiplies two fixed long integers and displays the result. The last task performs a binary search in a fixed list of 50 elements.

## 3. WCET

Now that we've implemented our tasks, we need to find the Worst Case Execution Time for our RTOS. To do this, we took the ‘python_execute.py’ code that you can find on our GitHub. With this code we ran each task 1000 times to find the maximum execution time for each one. Our total time is 0.26 seconds, but we need to add a factor of safety of a few milliseconds in order to get our WCET. In fact, adding 0.01 seconds for each task allows the system to simulate a WCET if we had run our code ad infinitum. This also allows us to have a pessimistic WCET and therefore avoid any problems. So our final WECT is 0.30 seconds. So we can find the period, which has to be greater than or equal to the WECT, so we could take 0.32 seconds to be sure that all the tasks can be executed.

## III. Conclusion

In conclusion, this project illustrates the significance and implementation of Real-Time Operating Systems for embedded systems and mission-critical applications. By utilizing FreeRTOS, we have developed a system capable of executing tasks with stringent timing requirements reliably. The methods employed, from program initialization to task creation and scheduling, ensure deterministic behavior and efficient resource utilization. Through the evaluation of Worst Case Execution Time, we ensure system robustness and mitigate potential risks. Overall, this project contributes to the understanding and practical implementation of RTOS principles, laying the foundation for the development of reliable real-time systems in various domains.