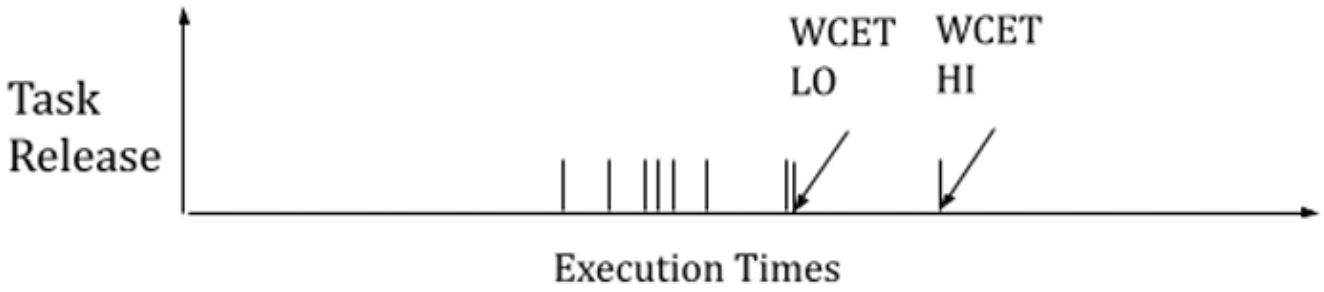
Managau Alice
Nicot Sébastien
4TS1

Real-Time Embedded Systems: Design of a RTOS



M. **SINGH** | Ln422

Abstract

In this project, we'll familiarize with the concept of the RTOS systems: these are the real-time operating systems on a machine.

Real-time operating systems (RTOS) play a crucial role in embedded systems, guaranteeing the execution of tasks on time. In this project, we designed an RTOS capable of handling various periodic tasks, doing a lot of different things. The system is composed of four periodic tasks to be designed: task 1 prints a status message indicating normal operation of our system. Task 2 is a function for converting degrees Fahrenheit to Celsius, always useful in physics and thermodynamics. Task 3 multiplies two large integers (*long int*) and displays the result. Finally, Task 4 performs a binary (or dichotomous) search on a predefined list.

To guarantee the system's efficiency and reliability, we carried out an analysis of task execution times and determined their longest execution times (also known as WCET). This decision was motivated by the need to conceive a predictable task planner with resource allocation, and thus, in fact the to define a time period. We then carried out a scheduling analysis to demonstrate the system's ability to meet task deadlines under different conditions.

By using the FreeRTOS platform, we implemented the task scheduler using priority orders. The scheduling algorithm ensures that tasks are executed according to their priority levels, with higher-priority tasks pre-empting lower-priority ones when it’s necessary.

This project contributes to the field of embedded systems by providing a practical example of RTOS design and scheduling analysis.

Introduction

Embedded systems play an essential role in many aspects of our daily lives, in many fields, and increasingly as technology advances. These applications can range from medical devices to autonomous vehicles, covering a wide variety of fields. To keep these systems running smoothly, it is crucial to use real-time operating systems (RTOS) capable of efficiently managing the time constraints and limited resources of certain devices and hardware.

This project aims to design and implement an RTOS capable of handling a variety of periodic tasks in an environment. The periodic tasks will enable us to perform a certain number of tasks, such as monitoring the state of the system and finding out whether it is working properly or not, by displaying a sentence. Another task would be temperature conversion or binary sorting of an ordered list of numbers, for example.

A thorough analysis of task execution times is carried out to ensure that the system can meet critical time constraints. The longest execution times (WCET) are determined for each task, which allows us an efficient and predictable resource planning. We will also prioritize tasks.

The practical implementation of this RTOS is carried out using the FreeRTOS platform, offering an open-source solution widely used in the field of embedded systems. Using the scheduling order, tasks are scheduled according to their priority, ensuring reliable and predictable system operation.

This project offers an opportunity to understand the challenges and considerations involved in designing embedded systems with real-time requirements. In addition, it provides a concrete example of RTOS implementation and scheduling analysis, contributing to the advancement of research in this critical area of embedded computing.

For this project we will proceed in 3 parts: First, we'll define the terms and choices we've made for this project. Next, we'll describe the tasks and their priorities. Finally, we'll analyze these tasks in terms of their temporal performance.

1. Definitions

Let's start by defining what a real-time operating system, also known as a RTOS. RTOSs are operating systems for which the maximum time (WCET) is an input and gives a specific output. These systems need a real clock to operate and schedule. The clock needs to be based on a real-world clock that measures the passage of time. This system is a multitasking system that schedules several tasks and execute them at the same time by scheduling them. The number of tasks executed simultaneously, and their priority are limited only by the hardware. It is one of the most widely used real-time operating systems on the market today, tried and tested.

These multitasking operating systems are used in real time in embedded systems such as cell phones or space systems. An RTOS is a system that allows us to schedule and control the execution of tasks: this enables the management of periodic (as in this work) as well as aperiodic and even sporadic tasks, while monitoring and guaranteeing deadlines. One of the positive points of RTOSs is that it allows us to implement preemption, which not all devices guarantee: preemption is the ability of an operating system to interrupt a task in progress to allow another to run, depending on priorities.

In our analysis, we'll be using FreeRTOS, which is a portable, preemptive, and open source RTOS. The device's portability prompted us to use it, but also its simplicity and compact form. Indeed, as we're carrying out a first project and a first approach to real-time information systems, we find it very suitable. Moreover, FreeRTOS provides us pre-prepared files suitable for microcontrollers, which is very convenient for us.

To go into a little more detail about how FreeRTOS scheduling works, we can define our task priorities by modifying or creating "*tskIDLE_PRIORITY*" constants. To modify these priority values, the request must be made directly by the programmer; there is no user request possible to modify this: we chose FreeRTOS for this because it allows us to control priorities and enable greater reliability.

2. Tasks description

To develop our project, we have a lot of tasks. In FreeRTOS, a task is a C function containing an infinite loop that execute our task periodically and does not return a result. Let's describe, more precisely, our tasks in the order of priority we give them and explaining why we give them this priority, and lets also define some concepts:

→ **Task 1: print « Working »**

This task is just a "control" task: this task allows users to know if the process is working well or not. We use firstly this task to check that everything is running correctly. In fact, this task is displayed every period in the console. This task is essential if we want to check that our code is viable and works correctly.

Thus, we've chosen to set this task as priority number 1, as it's here to check that everything's working well, so if it doesn't, the other tasks probably won't either. Displaying this data is the first step towards ensuring that the real-time system works properly, and that scheduling is carried out correctly.

To define this function, we define not only its priority, but also its syntax:

```
202  void task1(void * pvParameters)
203  {
204      TickType_t xNextWakeTime;
205      const TickType_t xBlockTime = mainTASK_SEND_FREQUENCY_MS;
206      const uint32_t ulValueToSend = mainVALUE_SENT_FROM_TASK;
207
208      /* Prevent the compiler warning about the unused parameter. */
209      ( void ) pvParameters;
210
211      /* Initialise xNextWakeTime - this only needs to be done once. */
212      xNextWakeTime = xTaskGetTickCount();
213
214
215
216      for ( ; ; )
217      {
218          vTaskDelayUntil( &xNextWakeTime, xBlockTime );
219          printf("Task 1 : Working");
220          printf("\n");
221
222      }
223  }
```

Figure 1: Task 1 syntax

We observe that the infinite *for* loop is defined in such a way that the condition is not specified. The task only stops at the end of the program. In this infinite loop, we implement a simple *printf* that is a function in C to display our message in the console. This kind of syntax, with an infinite loop, is widespread in RTOS systems.

This task alone shows the following output:

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ ./build/posix_de
mo

Trace started.
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Message received from software timer
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Task 1 : Working
Message received from software timer
```

Figure 2: Task 1 result in the console

We can see that our first task works well and print our “control” task.

→ **Task 2: Converting a fixed integer from fahrenheit to celsius**

First, let’s define the priority and why. We define this task as second priority. Indeed, the other tasks (others than the first one, described lately) don't necessarily need to have priority over each other (between tasks 2, 3 and 4). We decided to set the conversion task as second priority, as it requires the fewest resources than tasks 3 and 4.

This task has the following syntax, similar to before:

```c
void task2(void * pvParameters)
{
    TickType_t xNextWakeTime;
    const TickType_t xBlockTime = mainTASK_SEND_FREQUENCY_MS;
    const uint32_t ulValueToSend = mainVALUE_SENT_FROM_TASK;

    /* Prevent the compiler warning about the unused parameter. */
    ( void ) pvParameters;

    /* Initialise xNextWakeTime - this only needs to be done once. */
    xNextWakeTime = xTaskGetTickCount();

    float F = 75;
    float C;

    for ( ; ; )
    {
        vTaskDelayUntil( &xNextWakeTime, xBlockTime );
        C = (F - 32) / 1.8 ;
        printf("Task 2 : Convertion of %f (Fahrenheit) in %f (celsius)",F,C);
        printf("\n");
    }
}
```

Figure 3:Task 2 syntax

For this task, we've decided not to use the *scanf* function in C, as it would require the user to enter values. However, this method cannot be used in real time. By definition, It's not real time to ask the user to enter values. Instead, we define the values to be converted directly in the code. Here, we want to convert 75°F into degrees Celsius. This amounts to the following calculation: $T(°C) = \frac{(75°F-32)}{1.8} = 23.889°C$.

If we implement this task, following task 1 (and with a lower priority), we obtain the following result in the console:

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ ./build/posix_demo

Trace started.
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Message received from software timer
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
```

Figure 4:Task 2 result in the console.

We can see that the calculations verify the value calculated by the program. We can also see that the priority order of our tasks has worked well. Indeed, we can see that task 1 is executed before task 2. Our first task tells us that the program is running, and our second task performs a calculation.

→ Task 3: Multiplication of two integers

This task is the task we'll put on third priority. We've chosen to put it in this order because this task requires slightly more resources than the previous one. However, note that it is our order of priority because this task and the previous one can be exchanged. We'll put it in that order.

This task has the following syntax:

```c
void task3(void * pvParameters)
{
    TickType_t xNextWakeTime;
    const TickType_t xBlockTime = mainTASK_SEND_FREQUENCY_MS;
    const uint32_t ulValueToSend = mainVALUE_SENT_FROM_TASK;

    /* Prevent the compiler warning about the unused parameter. */
    ( void ) pvParameters;

    /* Initialise xNextWakeTime - this only needs to be done once. */
    xNextWakeTime = xTaskGetTickCount();

    long int A=256723412;
    long int B=198767312;
    long int multiplication;

    for ( ; ; )
    {
        vTaskDelayUntil( &xNextWakeTime, xBlockTime );
        multiplication=A*B;
        printf("Task 3 : Multiplication of %ld with %ld equal to %ld",A,B,multiplication);
        printf("\n");
    }
}
```

Figure 5:Task 3 syntax

For this task, we decide not to use the *scanf* function in c, for the same reasons specified above. Instead, we define the integers to be multiplied ourselves, in the c code. Note that *long int* takes 4 bytes for values between -2 147 483 648 up to 2 147 483 647. We then define $A = 256723412$ and et $B = 198767312$. With a scientific calculator, we'd get $A + B = 51\,028\,222\,530\,708\,544$. Let's check whether our program finds these values.

If we implement this task, following task 2 (and with a lower priority), we obtain the following result in the console:

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ ./build/posix_demo

Trace started.
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
```

Figure 6:Task 3 result in console

We can already see the order of priority that we defined earlier: task 1 is first, task 2 second and task 3 third. We can also see the result of multiplying the integers, and we can see that the result is good, given the calculation we've carried out with the calculator: our scheduling and our code seems to be working well so far.

→ **Task 4: Binary search in a list of 50 ordered items.**

In this task, we'll perform a binary search. First, let's define binary search sorting. Binary search - or dichotomous sort - applies to a sorted array (in ascending or descending order) and consists in continuously dividing the search interval. This reduces search time by having smaller intervals each time. This sorting method offers several advantages. It's more time-efficient, faster than other algorithms, and more efficient for large arrays (as we divide the search size in 2).

We'll set this binary sort task as the last priority. In our thoughts, this sort will take much longer than the other tasks, so we want to place it in last priority so as not to delay the other, less resource-intensive tasks.

We then have the following code:

```
void task4(void * pvParameters)
{
    TickType_t xNextWakeTime;
    const TickType_t xBlockTime = mainTASK_SEND_FREQUENCY_MS;
    const uint32_t ulValueToSend = mainVALUE_SENT_FROM_TASK;

    /* Prevent the compiler warning about the unused parameter. */
    ( void ) pvParameters;

    /* Initialise xNextWakeTime - this only needs to be done once. */
    xNextWakeTime = xTaskGetTickCount();
    int tableau[]={3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52}; //tableau de 50 éléments
    int n = sizeof(tableau) / sizeof(tableau[0]);
    int x=52;
    int low=0;
    int high=n-1;
    int check=-1;
    int m,milieu;


    for ( ; ; )
    {
        vTaskDelayUntil( &xNextWakeTime, xBlockTime );
        // jusqua ce que le pointeur haut et bas se rejoignent
        while (low <= high)
        {
            int milieu = low + (high - low) / 2;
            if (tableau[milieu] == x)
                //milieu=milieu;
                m = milieu;
                check =0;

            if (tableau[milieu] < x)
                low = milieu + 1;

            else
                high = milieu - 1;
        }


    if ( check == -1)
        printf("Task 4 : La valeur n'est pas dans le tableau \n");

    else
        printf("Task 4 : la valeur est à l'occurence  %d \n", m);
    }
}
```

Figure 7:Task 4 syntax

With this syntax, we can search for a value that is - or isn't - in the array. The program will tell us if the value is in the array, and at what index.

For example, let's look for the value 52 in the table, located at the last position, i.e. index 49 in the array defined in the syntax at figure 7.

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ ./build/posix_demo

Trace started.
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  49
```

Figure 8: Task 4 result in the console for the int 52

We can see the result of task 4, which is the last priority, and informs us that the value has been found at the desired occurrence. Let's test our program to see if it works for a value in the middle of the table, such as 26:

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ ./build/posix_demo

Trace started.
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
```

Figure 9:Task 4 result in the console for the int 26

We're working on a preemptive way here, as preemptive systems can stop the execution of one scheduled task in favor of another which takes priority. We do this because it gives us a guaranteed response. In fact, preemptive systems enable us to execute all our tasks on time and ahead of deadlines. Moreover, as mentioned above, preemptive scheduling optimizes resources by optimizing the different phases of a task (waiting, execution, etc.). CPU utilization is optimized.

Finally, to summarize our tasks and their priorities, we can draw up the following table:

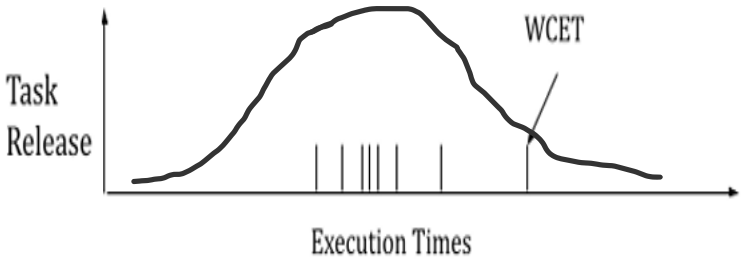
Task n°	Priority p_i
Task 1 (τ_1)	0
Task 2 (τ_2)	1
Task 3 (τ_3)	2
Task 4 (τ_4)	3

3. Tasks analysis

To continue this analysis, we'll try to analyze these tasks by their execution times. We do this because execution time is a good criterion for analyzing and comparing task performance, throughout time. To do this, we'll start by studying execution time statistics using the Linux *time* command. This command will give us statistics on a program's execution time. It is a good beginning to analyze tasks time because this command is included in all Linux system and do not require a specific program.

We then proceed in a different way to find the WCET for each task. WCET is defined as "worst case execution time": in fact, it's the longest time the program can take to run. It is important to define this in a multitasking system, as it enables tasks and periods to be scheduled. WCET is influenced by the hardware such as memory, processors, buses…

Obtaining a WCET is the most difficult part of designing an ideal process. Indeed, WCETs should not be too far from reality, but a margin must still be taken in case a task falls outside the average. Task execution times can be compared to a Gaussian curve:



The aim is to find the maximum value so that all the others are probably below this value. In fact, WCET is not deterministic, and it can vary on every execution. That is why we will run the code a lot of times to define the WCET, and we'll proceed in a pessimistic way: it means that we will take a WCET very far away from the average WCET. For that, we'll execute many times our programs.

Thus, firstly, we'll begin by analyzing the execution time statistics for each program. We'll do this using the command *time ./executable_name*.

When you apply this command, you get 3 pieces of information:

- "real": This indicates the time the task takes to run, from task start to task end. This includes execution time, but also other ancillary times such as waiting time before the execution in the processors, if another task is executing.
- "user": This indicates the user's total processing time. In other words, it's the time it takes the processor to execute the task.
- "sys": This is the total time used by the system. In other words, it's the time it takes the processor to execute the code and to execute system calls or kernel code.

We then have:

→ Task 1 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC/test_tasks$ time ./working
Task 1 : Working

real    0m0.011s
user    0m0.001s
sys     0m0.001s
```

→ Task 2 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC/test_tasks$ time ./celsius
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)

real    0m0.022s
user    0m0.002s
sys     0m0.009s
```

→ Task 3 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC/test_tasks$ time ./multiply
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544

real    0m0.028s
user    0m0.001s
sys     0m0.001s
```

→ Task 4 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC/test_tasks$ time ./binary
Task 4 : la valeur est à l'occurence  17

real    0m0.031s
user    0m0.000s
sys     0m0.002s
```

In each case, we'll use real time, which is the longest time and therefore true to the definition of a WCET: if we take into account the longest time, it will increase our WCET and suit our definition of a pessimistic WCET.

Secondly, to find the WCET for each task, we use a python program to run each task independently. This program is in charge of running each .c program a certain number of times to find its maximum execution time, in order to get the WCET in practice. Here, we'll run our programs 10000 times, to get a value that's high enough but not too unrealistic either, as said previously, it will suit a pessimistic WCET.

This code is expressed using the following syntax:

```python
import subprocess
import os
import tqdm

time_max=0
for i in tqdm.trange(10000):
    os.system("/usr/bin/time --output=outtime -p sh -c './working > /dev/null'")
    file=open("outtime","r")
    file.seek(0)
    f=file.readline()
    file.close()
    time=f[5:10]
    if float(time)>time_max:
        time_max=float(time)

print(time_max)
```

Figure 10: Python code to find the WCET of the first task ("working")

This gives us the following WCETs:

→ Task 1 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ python3 check_WCET.py
100%|████████████████████████████████████████████████████████████| 10000/10000 [01:25<00:00, 116.40it/s]
0.02
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$
```

We obtain $WCET_1 = 0.02s$.

→ Task 2 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ python3 check_WCET.py
100%|                                                                                | 10000/10000 [01:18<00:00, 127.68it/s]
0.01
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$
```

We obtain $WCET_2 = 0.01s$.

→ Task 3 :

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ python3 check_WCET.py
100%|                                                                                | 10000/10000 [01:17<00:00, 129.59it/s]
0.03
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$
```

We obtain $WCET_3 = 0.03s$.

→ Task 4 :

For this last task, we will test several cases, because our task can take different time, as it's a sorting algorithm.

- The value to be searched is the last in the list.

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ python3 check_WCET.py
100%|                                                                                | 10000/10000 [01:14<00:00, 134.37it/s]
0.01
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$
```

We obtain $WCET_4 = 0.01s$.

- The value to search for is the first in the list.

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ python3 check_WCET.py
100%|                                                                                | 10000/10000 [01:17<00:00, 129.26it/s]
0
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$
```

We obtain $WCET_4 = 0s$. Note that the values are truncated here, so we're probably very close to 0, but the last digits are truncated.

- The value to be searched for is embedded in the middle of the list.

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$ python3 check_WCET.py
100%|                                                                                | 10000/10000 [01:18<00:00, 128.00it/s]
0.01
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC$
```

We obtain $WCET_4 = 0.01s$.

- The value to be searched is not in the list.

```
alice@LAPTOP-COP0L52M:/mnt/c/Users/Alice/OneDrive/Documents/FreeRTOSv202107.00/FreeRTOS/Demo/Posix_GCC/test_tasks$ pytho
n3 check_WCET.py
100%|                                                                                | 10000/10000 [01:31<00:00, 109.57it/s]
0.03
```

We obtain $WCET_4 = 0.03s$

We can conclude that the location of the value to be searched for has little impact on the WCET of the task. We'll then take $WCET_4 = 0.03s$, the highest value ever encountered in our tests. We take the highest value for WCET to suit a pessimism way.

4. Period definition

Now that we have our WCET values for the four tasks, we can decide about the period of each task. In this case, we are free to decide on whatever period because our WCET are low and are similar (according to the different tasks), but we will take a period above the WCET on each task, so they can have time to run correctly. Note that, in real life and real projects, we don't really have this liberty.

Then, according to our task's priority, we decided to run with a higher frequency the task with higher priority. In this case, the task 1 will have the shortest period (and the higher frequency) and the task 4 the biggest one. Like this:

Period of each task:

- $T_1 = 500ms$
- $T_2 = 1s$
- $T_3 = 2{,}5s$
- $T_4 = 5s$

We can implement it on the code "*ipsa_sched.c*" file:

```
/* Priorities at which the tasks are created. */
#define mainQUEUE_RECEIVE_TASK_PRIORITY    ( tskIDLE_PRIORITY + 2 )
#define mainQUEUE_SEND_TASK_PRIORITY       ( tskIDLE_PRIORITY + 1 )

#define task1_TASK_PRIORITY                (tskIDLE_PRIORITY + 6 )
#define task2_TASK_PRIORITY                (tskIDLE_PRIORITY + 5 )
#define task3_TASK_PRIORITY                (tskIDLE_PRIORITY + 4 )
#define task4_TASK_PRIORITY                (tskIDLE_PRIORITY + 3 )

/* 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 )
#define task1_PERIOD_MS                    pdMS_TO_TICKS (500)
#define task2_PERIOD_MS                    pdMS_TO_TICKS (1000)
#define task3_PERIOD_MS                    pdMS_TO_TICKS (2500)
#define task4_PERIOD_MS                    pdMS_TO_TICKS (5000)
```

Figure 11: Priorities and period for each tasks

With this, we can replace the period on each task definition:

```

 void task1(void * pvParameters)
{
    TickType_t xNextWakeTime;
    const TickType_t xBlockTime = task1_PERIOD_MS;
    const uint32_t ulValueToSend = mainVALUE_SENT_FROM_TASK;

```

Figure 12: Definition of the period on task 1

Once done that, we can finally run our code with our finished scheduling (with the good period) and test again our program on Ubuntu, and see if our priorities and periods work well:

```
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 : Working
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Message received from software timer
Task 1 : Working
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Message received from software timer
Task 1 : Working
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 3 : Multiplication of 256723412 with 198767312 equal to 51028222530708544
Task 4 : la valeur est à l'occurence  23
Task 1 : Working
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Message received from software timer
Task 1 : Working
Task 1 : Working
Task 2 : Convertion of 75.000000 (Fahrenheit) in 23.888889 (celsius)
Task 1 : Working
```

Figure 13: 4 tasks results on the console, with good priorities and periods

With this result, we can see that everything worked well and that we have good priorities and periods for each task. Task 1 is working most of the time, then it's task 2 etc.…

None of the tasks are skipped or stopped, so our implementation is correct, and we do not have problems.

We've just realized a scheduled problem, with fixed priorities and preemptive, in FreeRTOS.

Conclusion

In this project we'd to design a RTOS for real-time embedded systems, and we explored the challenges and issues involved in designing systems capable of responding to real-time constraints. We tackled these issues through this practical work, which enabled us to discover the analysis of a real-time system. Embedded systems play an increasingly crucial role in many aspects of our daily lives, and the ability to effectively manage time constraints and limited resources is essential to ensure their smooth operation.

We firstly discussed the definition and operation of an RTOS, highlighting its ability to plan and schedule tasks efficiently, guaranteeing precise response times and maximizing the use of available resources. FreeRTOS was chosen as the development platform because of its portability, pre-emption, and open-source nature, providing a suitable solution for our project. Moreover, it's very convenient for beginners in RTOS thanks to pre-filled codes.

The practical design of the RTOS was carried out by implementing several periodic tasks, each with a priority defined according to its needs and the resources it uses. All tasks were organized and prioritized so as to put those consuming the least resources first, then we found the worst possible execution times (WCET) for each task [we put firstly those who take less resources, in our mind].

The pre-emptive approach adopted enabled efficient management of resources and a guarantee of response within strict deadlines, thus ensuring the smooth running of the system in demanding real-time environments.

In conclusion, this project has shed light on the fundamentals of RTOS design and implementation for real-time embedded systems, giving us an insight into this field.