HACLOSSIM

ALBERTO AMEGLIO[1], GIUSEPPE SALVEMINI[2], ENRICO DI STASIO[3], GIOVANNI LUCA DI BELLA [4]AND COSIMO VERGARI[5]*

Abstract. This article illustrates the main features of FreeRTOS, its implementation with QEMU, and various usage experiments aimed at deepening the main contents presented during the course. This project is based on embedded systems, which are now ubiquitous in modern technology, requiring specialized operating systems due to resource constraints and real-time requirements. For our project, we chose FreeRTOS, a lightweight, open-source real-time operating system. We will present the characteristics of this OS, an example tutorial on how to simulate an Arm architecture with QEMU, and explanations of the topics addressed and explored in depth by our group.

1. Introduction

Embedded systems permeate our daily lives, spanning from industrial automation to medical devices and the Internet of Things (IoT). These operating systems execute specialized tasks with constrained resources (memory, processing power, energy) and demand deterministic behavior to ensure real-time performance. Traditional operating systems, tailored for general computing, struggle within this constrained environment. The standout feature of FreeRTOS is its lightweight kernel, guaranteeing minimal resource consumption. Key characteristics include:

- **Lightweight design**: Utilizes minimal resources, making it optimal for embedded systems.
- **Open-source nature**: Community-driven innovation ensures constantly updated libraries.
- **Portability**: Adapts seamlessly to diverse hardware architectures, enhancing flexibility.
- **Task Scheduling**: FreeRTOS employs a priority-based queue to prioritize critical tasks, preempting less urgent ones. This deterministic approach ensures predictable execution times, crucial in the real-time realm of embedded systems.
- **Memory Management**: Operating within resource limitations, FreeRTOS employs dynamic allocation and memory pools to optimize memory usage.
- **Interrupt Management**: FreeRTOS swiftly responds to external events like sensor readings or button presses, enabling efficient handling.

However, FreeRTOS's true strength lies in its portability. It effortlessly adapts to various hardware architectures, rendering it a versatile operating system for myriad platforms. This adaptability is achieved by facilitating communication and data exchange among a wide array of devices[1]. For this project, the **Cortex-M MPS2** was selected, which is a multicore processor designed for high-performance embedded systems. The Cortex-M MPS2 finds widespread implementation in various fields such as industrial automation, motor control, robotics, communication systems, and medical devices. We will now provide a brief overview of QEMU, which is essential for our tutorial. QEMU is a free and robust open-source emulator that enables simulation of various hardware architectures, including those of embedded systems such as Arm-based devices. Unlike merely emulating the CPU, QEMU replicates the entire system, encompassing memory, peripherals, and other hardware components. This capability allows you to execute operating systems and applications within the virtual machine as if they were running on genuine hardware. QEMU stands out for its exceptional support for multiple Arm architectures, facilitating software development and testing across a broad spectrum of devices[5]. Its extensive customization options and scripting capability enable the creation of tailored simulations to meet specific requirements. Utilizing QEMU offers several advantages:

- Reduced development costs as physical hardware is not required.
- Increased testing flexibility, allowing the execution of multiple scenarios.
- Early debugging and problem identification.
- Safe experimentation without the risks associated with hardware manipulation.

Key words and phrases. Computer architecture, operating systems, OS embedded, FreeRTOS, QEMU, Cortex-M MPS2, Scheduling.

1.1. **Tutorial.**

(1) As the first point of our tutorial we need to download QEMU. Our base of support for the entire tutorial will be an Ubuntu 22.04.3 LTS operating system on Windows 10 x86. To download QEMU on the Ubuntu operating system, simply type the command: **sudo apt install qemu-system-arm**. The specification requires sudo to request the necessary privileges, apt is the software package manager for the Debian distribution in this case we will subsequently specify qemu -system-arm to explicitly request the extension for the arm architecture. To check if QEMU has been installed correctly on the machine we can use this command: **qemu-system-arm –version**, In our case the output is: QEMU emulator version 6.2.0.

(2) After that we will have to download FreeRTOS, available for free from the official documentation page [1] or from the official github repository[2].

(3) As the third points the fundamental element we need a debugger for Arm in this case we have selected the Arm GNU toolchain[3]. The GNU Arm toolchain (formerly known as the GNU Arm Embedded toolchain) is a collection of packages such as GCC (GNU Compiler Collection), Binutils, GDB, and others. It is used for developing embedded system software. This toolchain is intended for the 32-bit ARM Cortex-A, ARM Cortex-M, and ARN Cortex-R processor families. There is no easy way to download and determine the latest version of the toolchain via the command line. We then proceed by going to the original site and extracting the latest version of the toolchain as follows:

```
ARM_TOOLCHAIN_VERSION=$(curl -s https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads | grep -Po '<h4>Version \K.+(?=</h4>)')

```

Next, we download the archive file from the official website:

```
curl -Lo gcc-arm-none-eabi.tar.xz "https://developer.arm.com/-/media/Files/downloads/gnu/${ARM_TOOLCHAIN_VERSION}/binrel/arm-gnu-toolchain-${ARM_TOOLCHAIN_VERSION}-x86_64-arm-none-eabi.tar.xz"
```

You need to create a new directory to store the toolchain files: **sudo mkdir /opt/gcc-arm-none-eabi** We extract the toolchain files in the specified directory: **sudo tar xf gcc-arm-none-eabi.tar.xz –strip-components=1 -C /opt/gcc-arm-none-eabi** We add the /opt/gcc-arm-none-eabi/bin directory to the PATH environment variable:

```
echo 'export PATH=$PATH:/opt/gcc-arm-none-eabi/bin' | sudo tee -a /etc/profile.d/gcc-arm-none-eabi.sh"
```

To make the changes made effective, we can log out and log in to the system or run the following command to immediately apply the changes: **source /etc/profile**. We can check the version of the compilers: **arm-none-eabi-gcc –version arm-none-eabi-g++ –version**.Remove unnecessary archive file: **rm -rf gcc-arm-none-eabi.tar.xz**. On Linux, the Arm GNU toolchain provides GDB with Python support. Requires version 3.8 to be installed, any other version may return errors. To install the correct version of Python the commands are: **sudo add-apt-repository ppa:deadsnakes/ppa** and **sudo apt install python3.8**.

(4) At this point we can extract the FreeRTOS folder and we must access the directory taken into consideration: **cd : FreeRTOS/FreeRTOS/Demo** at this point we can delete all the other demos that we don't need except CORTEX_MPS2_QEMU_IAR_GCC. Now to run our simulation we need the command: **qemu-system-arm -machine mps2-an385 -cpu cortex-m3 -kernel build/gcc/output/RTOSDemo.out -serial stdio -s -S**

(5) To make changes to the OS we need to recreate the binary file for each change. To do this we need a make and cmake compiler[4]. Make is a tool widely used in Unix-like systems to automate the process of compiling programs. It is based on a configuration file called "Makefile", which contains instructions on how to compile and link source code into executables or libraries. For correct installation we must follow the following commands:

```
sudo apt-get install build-essential libssl-dev
cd /tmp
wget https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0.tar.gz
tar -zxvf cmake-3.20.0.tar.gz
cd cmake-3.20.0
./bootstrap
make
sudo make install
```

2. First Come First Served

First Come First Served is a non-preemptive scheduling algorithm where the waiting time of the single tasks depends on the time of its arrival.

Requirements

We need to adjust the config file of "FreeRTOSConfig.h" in the following way:

- set "configUSE_PREEMPTION" to 0, to deactivate the preemption since this is not a preemptive algorithm.

- set "configUSE_TIME_SLICING" to 0, to deactivate the time slicing feature of FreeRTOS. This feature is used to enable the round robin algorithm if some tasks have the same priority (as in this case).

2.1. **The code.** For the demo we used three tasks T1,T2,T3 arriving in this order at time 0, with the same priority ("mainTASK_PRIORITY = 2") and execution time of 1200ms, 300ms and 600ms respectively. Another task is also created for just printing the statistics called "statistics", with a lower priority.4.1

```
void main_fcfs( void ){
    xTaskCreate( task1, "T1", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY, NULL );
    xTaskCreate( task2, "T2", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY, NULL );
    xTaskCreate( task3, "T3", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY, NULL );
    xTaskCreate( statistics, "statistics", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY-1,
    NULL );
    vTaskStartScheduler();}
```

All the tasks do the same thing: they are just blocked in a loop until the right amount of CPU ticks are passed, and then they save their turnaround time and waiting time and terminate.

```
static void task1(void *pvParameters){
  TickType_t task1Start = xTaskGetTickCount();
  printf("Task1 started at time %dms\n", pdTICKS_TO_MS(task1Start));
  while (xTaskGetTickCount() - task1Start < pdMS_TO_TICKS(1200));
  TickType_t task1Stop = xTaskGetTickCount();
  printf("Task1 finished at time %dms\n", pdTICKS_TO_MS(task1Stop));
  wt1_fcfs = pdTICKS_TO_MS(task1Start);
  tt1_fcfs = pdTICKS_TO_MS(task1Stop);
  vTaskDelete(NULL);}
```

3. Shortest Job First

Shortest job first is a non-preemptive scheduling algorithm where the order of the tasks to be executed depends on their execution time: the shortest first.
Requirements Same as FCFS.2

3.1. **The code.** For the demo we used three tasks T1,T2,T3 arriving in this order at time 0, with different priorities based on their execution time:

- T1 with 1200ms execution time and priority set to 2
- T2 with 300ms execution time and priority set to 4
- T3 with 600ms execution time and priority set to 3

Another task is also created for just printing the statistics called "statistics", with a lower priority.4.1

```
void main_sjf( void ){
    xTaskCreate( task1, "T1", configMINIMAL_STACK_SIZE, NULL, TASK1_PRIORITY, NULL );
    xTaskCreate( task2, "T2", configMINIMAL_STACK_SIZE, NULL, TASK2_PRIORITY, NULL );
    xTaskCreate( task3, "T3", configMINIMAL_STACK_SIZE, NULL, TASK3_PRIORITY, NULL );
    xTaskCreate( statistics, "statistics", configMINIMAL_STACK_SIZE, NULL, STAT_PRIORITY, NULL );
    vTaskStartScheduler();}
```

4. Round Robin

Round-robin scheduling is a preemptive scheduling algorithm used in computing. It assigns time slices, also known as time quanta, to each process in equal portions and in a circular order, handling all processes without priority.
Requirements
Using a round robin algorithm on Freertos is easy because it's natively supported, we just need to make sure that all the tasks have the same priority and we need to modify the FreeRTOSConfig.h file and set configUSE_PREEMPTION to 1 and configUSE_TIME_SLICING to 1. It is also possible to specify the time quantum by modifying the configTICK_RATE_HZ, in fact a task is preempted at every tick and with that parameter we are setting the number of ticks in a second, for example if configTICK_RATE_HZ is set to 1000 it means that every 1ms we have a context switch. In our project we implemented round robin in a different way to better compute statistics like waiting time and turnaround time, we set configUSE_PREEMPTION to 1 but we disabled the time slicing to create a sort of handmade round robin.

4.1. **The code.** We created 3 tasks with the same priority that will be scheduled in round robin with a time quantum of 100ms, there is also a fourth task with a lower priority that will be used to compute the statistics and will be executed after the 3 tasks.

```
void main_rr( void ){
        xTaskCreate( task1, "T1", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY, NULL );
        xTaskCreate( task2, "T2", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY, NULL );
        xTaskCreate( task3, "T3", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY, NULL );
        xTaskCreate( statistics, "statistics", configMINIMAL_STACK_SIZE, NULL, mainTASK_PRIORITY
    -1, NULL );
        vTaskStartScheduler();}
```

The task1 lasts 1200ms, T2 lasts 300ms and T3 600ms. Each task is structured in the following way (in this case task1):

```
static void task1(void *pvParameters){
    TickType_t task1Start = xTaskGetTickCount();
    TickType_t task1Restart = task1Start;
    wt1_rr = task1Start;
    TickType_t task1Stop;
    printf("Task1 started at time %dms\n", pdTICKS_TO_MS(task1Start));
    while (xTaskGetTickCount() - wt1_rr < pdMS_TO_TICKS(1200)){
        if ((xTaskGetTickCount() - task1Restart) >= pdMS_TO_TICKS(100UL) && xTaskGetTickCount() -
    wt1_rr < pdMS_TO_TICKS(1200)){
            task1Stop = xTaskGetTickCount();
            printf("Task1 stopped at time %dms\n", pdTICKS_TO_MS(task1Stop));
            taskYIELD();
            task1Restart = xTaskGetTickCount();
            printf("Task1 restarted at time %dms\n", pdTICKS_TO_MS(task1Restart));
            wt1_rr += task1Restart - task1Stop;}}
    tt1_rr = xTaskGetTickCount();
    rt1_rr = task1Start;
    printf("Task1 finished at time %dms\n", pdTICKS_TO_MS(tt1_rr));
    vTaskDelete(NULL);}
```

The starting time of the task in ticks is saved in the variable task1Start, which is the time elapsed since the scheduler started. At this moment the waiting time is equal to task1Start so in the global variable wt1_rr is saved the same quantity. Then there is a while loop that will be interrupted only when the execution time of the task reaches 1200ms, the execution time is computed as the current time minus the time that the task spent in waiting state. Inside the while loop is checked if the task has been running for more than 100ms by computing the difference between the current time and the last time in which the task restarted, if this condition is true the stop time is saved in the variable task1Stop and a context switch is performed using the system call taskYIELD(). When the task restarts the restarting time is saved in the variable task1Restart and the difference between the restarting time and the stop time is added to the waiting time. At the end of the task the response time and the turnaround time are saved in the global variables. The job of the statistics task is printing the statistics of each task and compute the average waiting time, turnaround time and response time:

```
static void statistics(void *pvParameters){
    printf("Waiting time task1 = %dms, turnaround time = %dms, response time = %dms\n",
    pdTICKS_TO_MS(wt1_rr), pdTICKS_TO_MS(tt1_rr), pdTICKS_TO_MS(rt1_rr));
    printf("Waiting time task2 = %dms, turnaround time = %dms, response time = %dms\n",
    pdTICKS_TO_MS(wt2_rr), pdTICKS_TO_MS(tt2_rr), pdTICKS_TO_MS(rt2_rr));
    printf("Waiting time task3 = %dms, turnaround time = %dms, response time = %dms\n",
    pdTICKS_TO_MS(wt3_rr), pdTICKS_TO_MS(tt3_rr), pdTICKS_TO_MS(rt3_rr));
    printf("Average Waiting Time = %dms, average Turnaround Time = %dms, average Response Time = %
    dms\n", (wt1_rr+wt2_rr+wt3_rr)/3, (tt1_rr+tt2_rr+tt3_rr)/3, (rt1_rr+rt2_rr+rt3_rr)/3);
    vTaskDelete(NULL);}
```

5. TimeLine Scheduling

Timeline scheduling is a Real-Time non-preemptive scheduling algorithm used for periodic tasks. It consists of dividing the temporal axis in different slots of the same duration called minor cycles in which different tasks can be allocated for the execution. The duration of each minor cycle is defined by computing the greatest common divider (GCD) of all the task periods. The set of minor cycles forms the major cycle whose duration is define as the least common multiplier (LCM) of all the task periods. If all the tasks execute correctly within a major cycle, means that the scheduling is feasible.

Requirements

Since Timeline scheduling is not natively supported on FreeRTOS we created a simulation. It is based on 3 different tasks A, B and C executed in a major composed by 6 different minors. Each task has an execution time and a period defined using the

costants **mainEXEC_TASK_A (B, or C)** respectively 20 ms, 10 ms and 10 ms converted from ms to ticks using the function **pdMS_TO_TICKS()** and the constats **A(B or C)_period** respectively 40 ms, 80 ms and 120 ms. The major and minor tasks have been computed according to the period values of the tasks with the result of **mainTASK_Major_MS** (240ms) and **mainTASK_Minor_MS** (40ms). It's essential to modify **INCLUDE_vTaskDelay** bit to 1 in the FreeRTOSConfig.h file to be able to use the vTaskDelay function that will be used to simulate the minor duration. To measure the past time is used the function **xTaskGetTickCount()** that measures the number of ticks from the creation of the task Finally, since FreeRTOS aims to be lightweight and efficient, it does not include full support for floating-point operations by default, so to measure the CPU utilization factor it is essential use a division function implemented to represent the floating number as a string.

5.1. **The Code.** Starting from the main function we created the major task using the **xTaskCreate()** function assigning Major() as task_function, "major" name, configMINIMAL_STACK_SIZE as size of the stack (defined in FreeRTOSConfig.h and set to 800), configTASK_LOW_PRIORITY as priority value, with no need to pass any parameter to the function and no handler. Then the scheduler starts with the vTaskStartScheduler() function.

```
void mainTimeLineScheduling( void ){
BaseType_t major_task;
    major_task = xTaskCreate( Major,
                  "Major",
                  configMINIMAL_STACK_SIZE,
                  NULL,
                  configTASK_LOW_PRIORITY,
                  NULL);
    vTaskStartScheduler();}
```

The major function is the core of the algorithm itself; it initializes 3 main variables:

- StartTime → used as reference for the beginning of the task.
- xBlockTime → used to measure the execution time of the relative minor and will be used to compute the remaining time the minor will be put waiting.
- $CPU_{utilization-factor}$ → used to compute the cpu utilization factor computed as:

The $CPU_{utilization-factor}$ is initialized to zero at the beginning of the major task, then the Minor1() is executed returning the time referred to the beginning of the execution. The StartTime is used to compute the execution time of the minor as follow

xBlockTime= xTaskGetTickCount()-StartTime

Then a conditional statement is performed, checking whether the execution time of the minor is minor than the execution interval associated to it. If yes, then is called the vTaskDelay() function passing the remaining time (mainTASK_Minor_MS – xBlockTime) as parameter.

```
void Major(){
    TickType_t xNextWakeTime;
    uint32_t xBlockTime;
    uint32_t StartTime = 0;
    uint32_t CPU_UTILIZATION_FACTOR;
    while(1){
        CPU_UTILIZATION_FACTOR = 0;
        StartTime = Minor1();
        xBlockTime = xTaskGetTickCount() - StartTime;
        if(xBlockTime <  mainTASK_Minor_MS)
            vTaskDelay(mainTASK_Minor_MS - xBlockTime);
        CPU_UTILIZATION_FACTOR += xBlockTime;}
```

After the delay, is added to the cpu utilization factor variable the execution time of the minor, that will be used at the end of the major to measure how much the CPU has been out of idle.

$$CPU_{utilization} = \frac{CPU_{utilization-factor}}{mainTASK_{Major_{ms}}}$$

Where:

- $CPU_{utilization-factor} = \sum xBlockTime$ (meaning the execution time of each Minor)
- mainTASK_Major_ms = as defined before is a costant with value 240 ms

```
Void Major(){
   printf("\nCpu_Utilization_factor : %s\n", division(CPU_UTILIZATION_FACTOR, mainTASK_Major_MS,
    3)); //precision = decimal numbers
        break;}
    vTaskDelete(NULL);
```

Since this is a simulation, after the execution of the minor, using a break the while loop is left and the task is deleted using the **vTaskDelete()**. The structure of the minors is similar for all, just changes which task is executed, according to their period and execution time. Each minor prints the time it starts executing.

```
void Minor4(){
    TickType_t xNextWakeTime = xTaskGetTickCount();
    printf("\nMinor 4: %d\n", pdTICKS_TO_MS( xNextWakeTime ));
    A(0);
    C();}
void Minor5(){
    TickType_t xNextWakeTime = xTaskGetTickCount();
    printf("\nMinor 5: %d\n", pdTICKS_TO_MS( xNextWakeTime ));
    A(0);
    B();}
```

Each task prints the time it starts executing, and the time it terminates its execution. To simulate the execution time is created a polling loop that looks for the condition:
$xTaskGetTickCount() - StartTime < mainExec_{task}$

```
TickType_t A(TickType_t Start){
    TickType_t StartTime = xTaskGetTickCount() - Start;
    printf("A started at: %d\n", StartTime );
    while(xTaskGetTickCount() - StartTime < mainEXEC_TASK_A);
    printf("  finished at: %d\n", pdTICKS_TO_MS( xTaskGetTickCount() ));
    return StartTime;}
```

6. Rate Monotonic

Rate Monotonic is a preemptive scheduling algorithm for periodic tasks where each task is given a static priority based on the inverse of its period. This allows for task with smaller periods (more frequent) to execute first reducing the risk of missing their timeline. **Requirements**
We activated preemption like we did for other algorithms requiring it by setting configUSE_PREEMPTION to 1 in the FreeRTOSConfig.h file. We also modified the minimal stack size value to 800 words by setting the configMINIMAL_STACK_SIZE to 800, the total heap size to 120 KiloBytes by setting the configTOTAL_HEAP_SIZE to 120 * 1024.

6.1. **The code.** For the demo we used three tasks similar to those used in the example in the slides having respectively: Periods T1=40ms, T2=80ms, T3=120ms and Worst Case Execution Times WCET1=20ms, WCET2=10ms, WCET3=10ms. We also used a 4th one to mimic the idle CPU task to allow us to track the percentage of time actually spent executing the tasks more easily. For each of the three tasks we created also a corresponding timer to make them periodic, each timer starts at zero (the tasks all arrive at time 0) and each one has a duration corresponding to the task's period. After a timer reaches the value set it creates a new instance of the corresponding task.

```
void main_RM( void ){
    xTaskCreate( task1, "T1", configMINIMAL_STACK_SIZE, NULL, TASK1_PRIORITY, NULL );
    xTaskCreate( task2, "T2", configMINIMAL_STACK_SIZE, NULL, TASK2_PRIORITY, NULL );
    xTaskCreate( task3, "T3", configMINIMAL_STACK_SIZE, NULL, TASK3_PRIORITY, NULL );
    handle = xTaskCreate( taskIdle, "TIdle", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY,
    NULL );
    TimerHandle_t timer1 = xTimerCreate("Timer1", pdMS_TO_TICKS(40), pdTRUE, NULL, timer1func);
    xTimerStart(timer1, 0);
    TimerHandle_t timer2 = xTimerCreate("Timer2", pdMS_TO_TICKS(80), pdTRUE, NULL, timer2func);
    xTimerStart(timer2, 0);
    TimerHandle_t timer3 = xTimerCreate("Timer3", pdMS_TO_TICKS(120), pdTRUE, NULL, timer3func);
    xTimerStart(timer3, 0);
    TimerHandle_t timer4 = xTimerCreate("Timer4", pdMS_TO_TICKS(5000), pdTRUE, NULL, timer4func);
    xTimerStart(timer4, 0);
    vTaskStartScheduler();}
```

We also created a 4th timer with duration of 5000ms (5 second) to put an end to the demo and to run a function that calculates the CPU Utilization Rate. Since the priorities are static we simply define the task with the shortest period to have the highest priority value (TASK1_PRIORITY = tskIDLE_PRIORITY +3) that corresponds to the highest priority in FreeRTOS. The tasks are very simple, they are implemented using a for cycle that approximates the duration with a counter common to all tasks to be later used for the utilization factor, and use the system call xTaskGetTickCount() to display the time at which the task starts and finishes.

```
static void timer4func(void* parameters){
    int temp = CPUIdle+CPUnotIdle;
    printf("CPU Idle count %d\n",CPUIdle);
    printf("CPU inTask count %d\n",CPUnotIdle);
    printf("CPU Utilization Rate around %s", division(CPUnotIdle, temp, 3));
    while(1);}
    char* division(int num, int den, int prec) {
    int Intpart = num / den;
    long long int rest = num % den;
    char* res = (char) pvPortMalloc(prec + 3);
    sprintf(res, "%d,", Intpart);
    for (int i = 0; i < prec; i++) {
        rest *= 10;
        long long int cipher = rest / den;
        rest %= den;
        res[i+2] = '0' + cipher;
    }
    res[prec+2] = '\0';
    return res;}
```

In the upper code there is the calculation of the CPU Utilization Rate, where we implemented a function that doesn't require the use of floating point numbers (not natively supported by FreeRTOS), in the lower one the one for a generic task and timer-activated function (task1 and timer1 function in this case).

```
static void task1(void *pvParameters){
TickType_t task1Start = xTaskGetTickCount();
printf("Task1 started at %d ms\n", pdTICKS_TO_MS(task1Start));
for (int i=0; i<700000; i++){
CPUnotIdle+=1;}
printf("Task1 finished at %d ms\n", pdTICKS_TO_MS(xTaskGetTickCount()));
vTaskDelete(NULL);}
```

Beside displaying the CPU Utilization Rate, the function linked to timer 4 also starts a infinite while loop to stop the execution of other tasks after the 5 seconds mark

References

1. FreeRTOS™ Real-time operating system for microcontrollers https://www.freertos.org/index.html
2. GitHub FreeRTOS https://github.com/FreeRTOS/FreeRTOS
3. Install Arm GNU Toolchain on Ubuntu 22.04. https://lindevs.com/install-arm-gnu-toolchain-on-ubuntu
4. How to Install CMake on Ubuntu 22.04 LTS https://vitux.com/how-to-install-cmake-on-ubuntu/
5. QEMU Documentation. https://www.qemu.org/

[1] Master's Degree Course in CYBERSECURITY at the Polytechnic of Turin teaching class: LM-32 (DM270))
Email address: s330946@studenti.polito.it

[2] Master's Degree Course in CYBERSECURITY at the Polytechnic of Turin teaching class: LM-32 (DM270))
Email address: s322886@studenti.polito.it

[3] Master's Degree Course in CYBERSECURITY at the Polytechnic of Turin teaching class: LM-32 (DM270))
Email address: s323075@studenti.polito.it

[4] Master's Degree Course in CYBERSECURITY at the Polytechnic of Turin teaching class: LM-32 (DM270))
Email address: s332088@studenti.polito.it

[5] Master's Degree Course in CYBERSECURITY at the Polytechnic of Turin teaching class: LM-32 (DM270))
Email address: s329479@studenti.polito.it