Academic Year 2023/2024

Computer Architecture & Operating Systems

HacklOSsim Project

Project Report

Group 20

Sannia Gabriele - S331385
Sabella Mattia Luigi - S285363
Pittalis Domenico - S283602
Nobili Luca - S331461

Contents

1 Introduction to FreeRTOS

FreeRTOS is a market-leading **Real-Time Operating System** (**RTOS**) for microcontrollers that was released after 18 years of work in partnership with the world's leading chip companies. Now FreeRTOS is downloaded once every 170 seconds and is distributed freely under the MIT open source license. FreeRTOS includes a kernel and a set of libraries suitable for use across all industry sectors.

FreeRTOS provides robustness, tiny footprint, detailed pre-configured demos, Internet of Things (IoT) reference integrations and scalable size. In addition, FreeRTOS has a minimal ROM, RAM and processing overhead and it is very simple as the core of the RTOS kernel is contained in only three C files. FreeRTOS provides us with the following **functionalities**:

1.1 Tasks

A real time application that uses a RTOS can be structured as a set of **independent Tasks**. Since FreeRTOS is designed as a **single-processor OS**, **only one** task within the application can be at the Running state at any point in time and the **Real-Time Scheduler** is responsible for deciding which task this should be. As tasks have **no knowledge** of the scheduler activity, it is its responsibility to ensure that the **Context Switch** is performed correctly. To achieve this, each task is provided with its own **Stack**. Thus, when the task is **swapped out**, its execution context is saved to the stack of that task so it can also be restored when the same task is later **swapped in** to continue its execution.

Every time a task is created, a priority level, ranging from **0** to (**configMAX_PRIORITIES - 1**) **must** be assigned to it. Lower values indicate tasks with lower priority, thus, the **IDLE Task** has **always** assigned a priority of **0** (**tskIDLE_PRIORITY**). The scheduler **must** ensure that tasks in the Ready or Running state always receive CPU time preference over tasks with lower priority that are also in the Ready state. In simpler terms, the task transitioning to the Running state is **always** the highest-priority task eligible for execution. However, multiple tasks can share the same priority level. Thus, if **configUSE_TIME_SLICING** is either **undefined** or set to **1**, Ready state tasks with identical priorities will distribute the available processing time using a **Time-Sliced Round-Robin Scheduling Algorithm**.

1.2 Semaphores

FreeRTOS provides **Binary Semaphores** and **Counting Semaphores**. While Binary Semaphores and **Mutexes** are similar, they differ subtly: Mutexes feature a priority inheritance mechanism, whereas Binary Semaphores do **not**. Consequently, Binary Semaphores are preferable for **synchronization tasks**, while Mutexes excel at **simple mutual exclusion tasks**.

FreeRTOS's Semaphore API functions allow specification of a block time, indicating the maximum duration a task should remain in a Blocked state when attempting to acquire a semaphore. If multiple tasks block on the same semaphore, the highest priority task will be unblocked first when the semaphore becomes available.

Similar to how Binary Semaphores represent Queues of length one, while Counting Semaphores can be visualized as Queues with a length greater than one and are primarily used for:

- **Counting Events**: handlers *'give'* a semaphore for each event occurrence, while tasks *'take'* a semaphore for each event processing;
- **Resource Management**: the count value denotes available resources. Tasks **must** *'take'* a semaphore by decrementing the count value to obtain control of a resource. When the count reaches **0**, no free resources are available. Tasks *'give'* back the semaphore upon resource release, incrementing the count value. It is preferable for the count value to be equal to the maximum count value at semaphore creation;

1.3 Queues

Queues are fundamental for enabling communication between tasks in a multitasking system. They serve as channels through which tasks can exchange messages, facilitating coordination and data sharing. Typically, Queues operate on a first-in-first-out (FIFO) basis, meaning that the data sent earlier is received first. However, it is also possible to send data to the front of the Queue if necessary. Queues can handle both small messages stored directly in C variables and larger messages by passing pointers. This approach also supports variable-sized messages and enables a single queue to receive different message types. Queues can block tasks when they are full or empty, allowing for efficient resource utilization. Overall, FreeRTOS queues provide an intuitive and efficient mechanism for inter-task communication in real-time embedded systems.

1.4 Event Bits & Event Groups

Event Bits (or **Event Flags**) are used to indicate whether an event has occurred or not. They can be defined as individual bits that take on a specific value to represent the state of an event. For example, an application might define a bit that indicates whether a message has been received and is ready to be processed (1) or if there are no messages waiting to be processed (0).

Event Groups are sets of event bits, where each bit is identified by a bit number. For instance, an event group might include bits representing different states of the application, such as message reception, queuing a message to send to the network, or sending a heartbeat message on the network. The API functions for event groups allow tasks to set, clear, and wait for one or more event bits to be set within an event group. They can also be used to synchronize tasks, creating synchronization points where tasks wait until all other involved tasks have reached their synchronization points.

2 Examples

2.1 Basic Tasks: *"semaphoresAcyclic.c"*

This example uses a Counting Semaphore (sem1) that is created with max value 2 and initial value 0 in order to allow task1 to be executed before task2 and task3, a Binary Semaphore (sem2) and a Mutex (Mutex) to protect the critical section (stdout). The purpose of the exercise is to accomplish the following Acyclic Semaphore Precedence Graph.

task1, which does **not** need to synchronize with any other task, releases semaphore sem1 twice after printing, thereby unblocking task2 and task3, both of which subsequently print and release sem2 (Note that just one semaphore to unlock 2 task because the graph is acyclic). Now, task4 (which was supposed to acquire sem2 twice) can print and terminate.

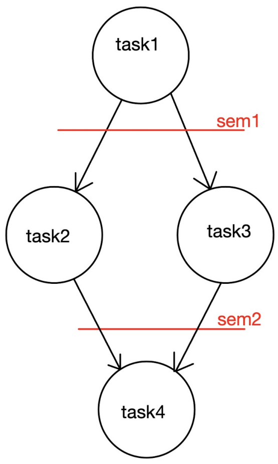

Figure 1: Acyclic Semaphore Precedence Graph

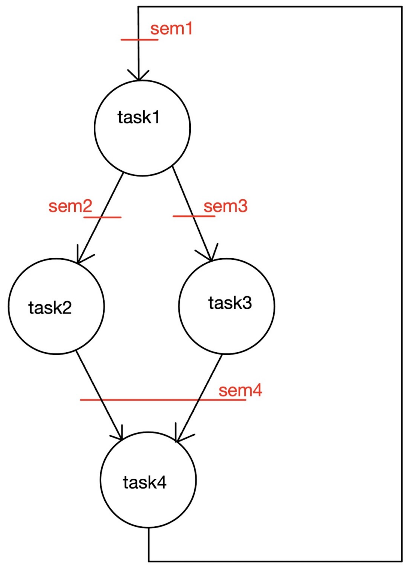

Figure 2: Cyclic Semaphore Precedence Graph

2.2 Extended Tasks: *"semaphoresCyclic.c"*

The substantial difference from the previous example is that here **Extended Tasks** are implemented, leading to the creation of a Cyclic Semaphore Precedence Graph. However, it is not possible for one task to unblock two other tasks using the same semaphore, for this reason task2 and task3 require two independent semaphores (if they were to share the same semaphore, it could happen that one of the tasks acquires it twice while the other does not acquire it at all).

Another peculiarity to note is that task1 (being the first to execute) is the only task whose semaphore (another even more than the previous example) has a value of 1 upon its creation; otherwise, the tasks would all remain permanently blocked.

2.3 Queue: "*queueChar.c*"

In the ***"queueChar.c"*** example script, our aim is to demonstrate how FreeRTOS handles two tasks that try to work **concurrently** using Queues. The code creates two tasks: **queueSendTask** and **queueReceiveTask**, both with the same priority, which send and receive characters via a queue, for this reason the Round-Robin is used.

The **queueSendTask** task sends characters from a string defined as global to the queue one by one at regular intervals of 200 milliseconds.

The **queueReceiveTask** task receives characters from the Queue and prints them to the console, with a delay of 500 milliseconds between each received character.

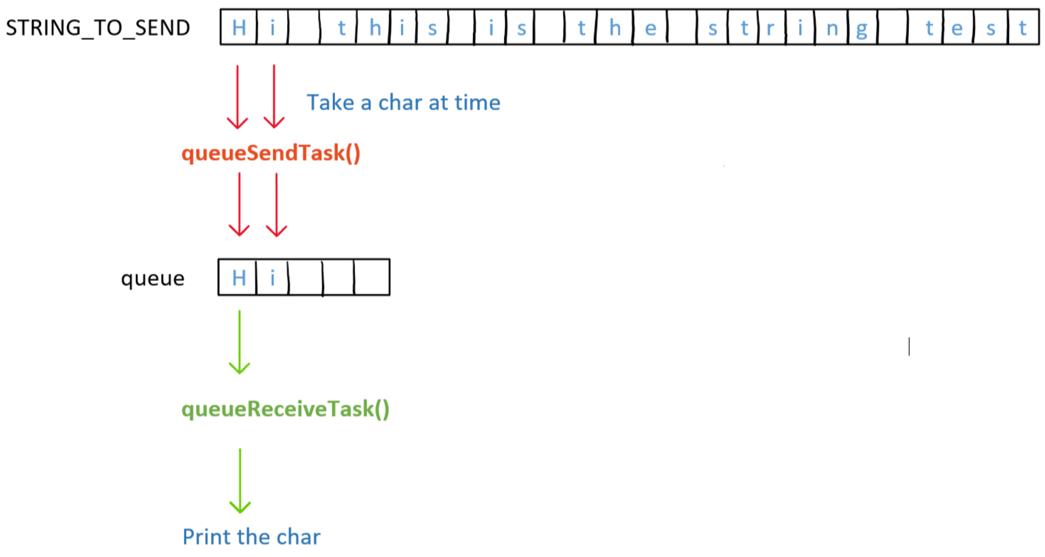

Figure 3: Queue Example

2.4 Event Group: "*event2Tasks.c*"

The example *"event2Tasks.c"* shows how to coordinate the concurrent execution of different tasks using **event-Groups**. The code defines three main tasks: **Task1**, **Task2**, and **Task3**.

Task1 takes care of the initialization of the system. Essentially, it creates an event group (**eventGroup**) that is used to coordinate the execution of tasks. **Task1** then creates **Task2** and **Task3** that alternate their execution. Initially, **Task1** notifies **Task2** to start it. Once this is done, **Task1** finishes executing.

Task2 and **Task3** form the core of multitasking. They wait for the turn event respectively and when it is received, they perform their specific task (printing a message in the specific case). After completing their task, each task notifies the next task in the loop of the event so that it can execute.

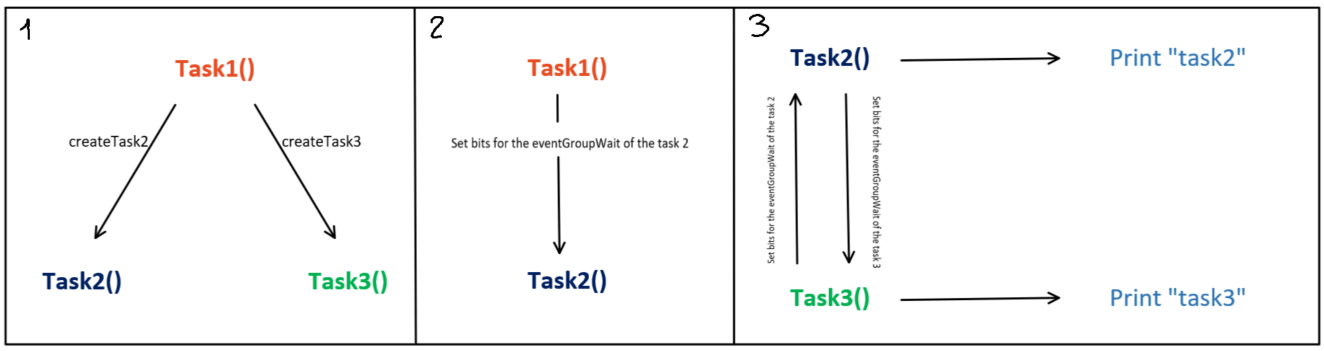
1
Task1()
createTask2
createTask3
Task2()
Task3()
2
Task1()
Set bits for the eventGroupWait of the task 2
Task2()
3
Task2()
Print "task2"
Set bits for the eventGroupWait of the task 2
Set bits for the eventGroupWait of the task 3
Task3()
Print "task3"

Figure 4: EventGroup Example

3 EDF Scheduler

The **Earliest Deadline First** (**EDF**) algorithm is a **scheduling policy** used in real-time systems to schedule tasks based on their deadlines. EDF assigns priorities to tasks **dynamically**, with the task having the earliest deadline being given the highest priority. This ensures that tasks with closer deadlines are executed first, reducing the likelihood of missing deadlines. Additionally, EDF uses **preemption** which ensures that higher-priority tasks can interrupt the execution of lower-priority tasks when necessary

In implementing the EDF algorithm, our **primary objectives** were to maintain FreeRTOS's original structure as possible and handle aperiodic tasks as **soft real-time systems** with a **Background Scheduling** approach. This entails scheduling aperiodic tasks based on their arrival order, ensuring execution when no other periodic tasks are in the ready list. Furthermore, ensuring system feasibility is crucial to enable the scheduling algorithm to operate effectively.

All the changes we are about to illustrate pertain to the *"tasks.c"* file. To allow a user to select among the various scheduling algorithms of FreeRTOS, including the one we propose, we added **configUSE_EDF** to *"FreeRTOSConfig.h"*. By setting it to **1**, the EDF scheduler will be used.

As mentioned earlier, our scheduling requires knowledge of the **deadlines** for each task in order to schedule them correctly. For this reason, a proper parameter has been added to the Task Control Block (TCB). Therefore, a new task creation function that allows passing the deadline as a parameter is necessary.

The new function is called **xTaskCreateEDF** and, as previously mentioned, it accepts a parameter of type **TickType_t** for the deadline. Additionally, within it, the value is saved through a pointer to the TCB, which will be used later to calculate the priority associated with the task. Our EDF structure entails that the priority (hence the deadline) of a task resides in the *Item* linked to the **xStateListItem**. Thus, using a pre-implemented function, **listSET_LIST_ITEM_VALUE**, we set the current tick as the initial value, which before calling **vTaskStartScheduler** will be **0**. The **xStateListItem** is a data structure used internally by FreeRTOS to manage task state information. It is a member of the TCB. By utilizing **xStateListItem**, containing information about the task's state and its reference to the state list, we can dynamically track the task's priority.

FreeRTOS normally uses multiple lists to manage tasks. However, for our scheduler, we prefer **simplicity**. So, we've decided to use just one list. This list will arrange tasks based on their deadlines (the **listItem** in **xStateListItem** mentioned before), with the closest deadline at the top. To make this happen, we've created a new list called **xReadyTasksListsEDF**.

With the changes made so far, we've successfully created a task that accepts the deadline as a parameter, saves it in the TCB, and sets it as the initial parameter of the **xStateListItem**. Additionally, we've created the new ready list.

The next modification deals with how deadlines and priorities are updated. To achieve this, we've modified the **prvAddTaskToReadyList** function, responsible for inserting tasks into the ready list, in order to update the deadline every time a task needed to be inserted into the ready list. Firstly, we calculate the new deadline by adding the value of the previous deadline contained in the **xStateListItem**'s item to the declared deadline in the task creation, which had been saved in the TCB.Subsequently, we set the new value of the **xStateListItem**'s item using **listSET_LIST_ITEM_VALUE**. Finally, we call the **vListInsert** function (originally from FreeRTOS), which positions our task within the ready list according to the ascending order of deadlines.

The management of **IDLE tasks** is also revised. Initialization of the IDLE task occurs within the **vTaskStartScheduler** method.

In the standard FreeRTOS scheduler, the IDLE task is a straightforward task initialized with the lowest priority. Consequently, it is scheduled **only** when no other tasks are ready. However, with the EDF scheduler, the lowest priority behavior can be simulated by assigning a task with the farthest deadline. Therefore, a high value (e.g. **1000000**) is assigned as the deadline.

The next required adjustment involves the **context switching mechanism**. In the function **vTaskSwitchContext** with the FreeRTOS scheduler, a dedicated function was used to retrieve the highest priority task. With the new ready list, it will be sufficient to fetch the task at the head of it using the function **listGET_OWNER_OF_HEAD_ENTRY** which is already implemented in FreeRTOS, as it is used by other functions that work on lists like ours for EDF (e.g. delayed List). Given the memory address of the ready list, this function returns the TCB of the new task to be executed.

To enable preemption, an additional step is necessary. We need to specify when the context switching is required after a task returns to the ready list (for example, when it exits from the delayed list associated with **vTaskDelay**). Therefore, in the **xTaskIncrementTick** task, for each tick, it is checked whether a task returns to the ready list or not. Consequently, the deadlines of the new task inserted into the ready list are compared with those of the task in execution (the deadlines are contained in the **xStateListItem** explained earlier). If the deadline of the task just reinserted into the ready list is strictly less than that of the task in the running state, the variable **xSwitchRequired** is set to **True**, indicating that a context switch is necessary. The decision to use *'strictly less than'* rather than *'less than or equal to'* is based on the fact that it reduces the number of context switches performed. It is easy to see that if two tasks have the **same deadline**, it is inefficient to execute a swap of the running state.

To manage the aperiodic tasks, it was decided **not** to use a new ready list. Instead, to allow for a **simpler** and **more intuitive** structure, the same ready list as the periodic tasks is utilized. In fact, a **Background Scheduling** based on the arrival order is created by placing these tasks slightly below the fictitious deadline of the IDLE task. This ensures that they are executed when there are no other periodic tasks. Therefore, a dedicated **vTaskCreate** function has been created for them, where their "deadline" is set following a counter that defines their arrival order (**taskIDLEdeadline - count**). This means that the number of aperiodic tasks is limited to the value of the counter. For this reason, it was chosen to allow the user to set this parameter in the ***"FreeRTOSConfig.h"*** file.

3.1 Performance Evaluation & Comparison

To demonstrate the functionality of the EDF scheduler with aperiodic tasks, the ***"main_edf.c"*** demo was created containing two periodic tasks and two aperiodic tasks, assuming coinciding periods and deadlines. The two periodic tasks, **TaskA** and **TaskB**, have periods of 40 and 70 ticks respectively, with Worst-Case Execution Times (WCET) of 20 and 30. The aperiodic tasks, **TaskC** and **TaskD**, have a WCET of 10. To demonstrate their correctness, the software Cheddar was used. Cheddar is a GNU GPL real-time scheduling simulator and schedulability tool. The *"main_edf_aperiodic.c"* demo with **only** periodic tasks is also provided.

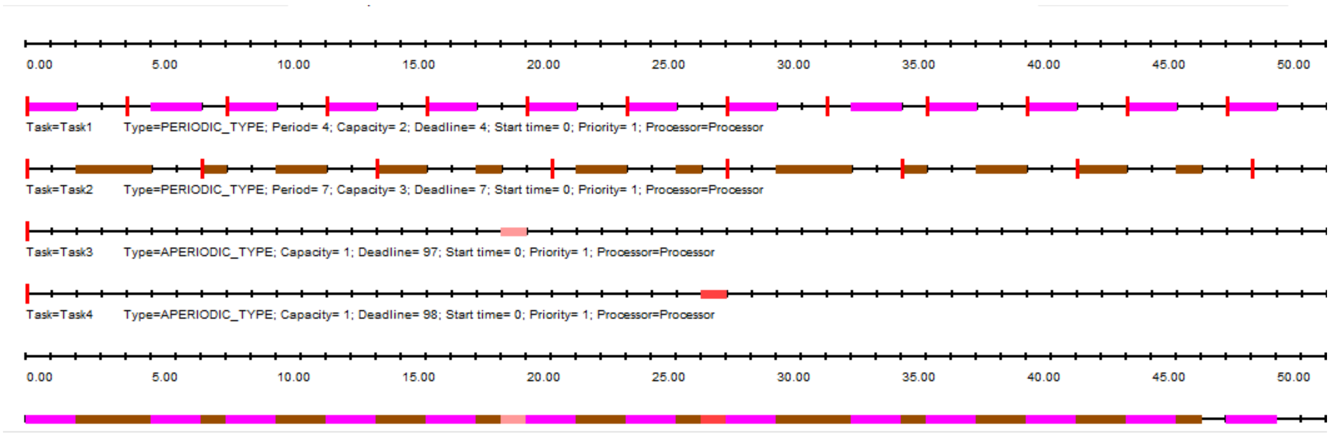

Figure 5: EDF Scheduler Performance Evaluation

4 Multilevel Queue Scheduler

The implemented **Multilevel Queue Scheduler** utilizes the five Ready Lists of the basic FreeRTOS scheduler, where each list is associated with a different priority: the first four lists, including the one containing the IDLE task, operate based their priorities with preemptive scheduling and **without** time slicing. While the fifth list, which has the highest priority, works using Round-Robin.

Thus, a Context Switch can happen **only if one or more** of the following conditions are true:

- A higher priority task enters the Ready Lists;
- The Running Task is suspended (which means it is removed from the its Ready List);
- The time-slice of the task is finished;

When a task is suspended and exits its Ready List, its initial priority is restored. To achieve this, the **UBaseType_t uxOldPriority** variable is added to the TCB to store the old priority of the task. On top of that, the **int cc** variable is also added to the TCB, but it has **only** an internal purpose.

The **xTaskIncrementTick** function has been modified in order to:

- Implement round-robin on the highest priority list, which is activated **only** when there are **at least** two task inside it;
- Restore the priority of a task when it is reinserted into the appropriate Ready List;
- Mitigate **Starvation** of low priority task through the implementation of an **Aging** mechanism: every **tick_update_rate** ticks, all tasks (except the running one) in the first four Ready Lists will get their priority increased by **1**, until they reach the highest priority list.

  This increment is operated, each time the system tick counter reaches a multiple of the **tick_update_rate**, by the **xTaskPrioritySet** function (a modified version of the **vTaskPrioritySet** function);

4.1 Performance Evaluation & Comparison

In the *"main_prova_mlq.c"* demo, two tasks are used: **Task1** with priority **3**, which is **never** suspended, and **Task2** with priority **1**. At the end of its execution, **Task2** is suspended for 10 ticks and then placed back in the Ready List, for three times. This demo allows us to show:

- How priority is increased every time that the system tick is a multiple of the **tick_update_rate**;
- The restoration of the old priority of **Task2** once it is re-inserted in the Ready List;
- The activation of round-robin when both tasks are in the highest priority list;

5 Work Contributions

1. Gain proficiency in using QEMU for running an embedded operating system and create a tutorial detailing the installation and usage procedures:
   - General understanding of the installation and usage procedures of QEMU and FreeRTOS:
     - Sannia Gabriele - S331385
     - Sabella Mattia Luigi - S285363
     - Pittalis Domenico - S283602
     - Nobili Luca - S331461
   - Creation of the ***Step-by-Step Installation Guide.pdf***:
     - Sannia Gabriele - S331385
     - Pittalis Domenico - S283602
2. Develop practical examples/exercises demonstrating the functionality of the simulator/operating system in alignment with the topics studied in class:
   - General understanding of the main FreeRTOS functionalities (e.g. task creation and management):
     - Sannia Gabriele - S331385
     - Sabella Mattia Luigi - S285363
     - Pittalis Domenico - S283602
     - Nobili Luca - S331461
   - Creation of the ***Project Presentation group20.pptx*** and ***HacklOSsim Report.pdf*** relating the main FreeRTOS functionalities:
     - Sannia Gabriele - S331385
     - Sabella Mattia Luigi - S285363
3. Customize the operating system to implement a new solution (e.g., scheduling, memory management, etc.):
   - EDF Scheduler implementation:
     - Sabella Mattia Luigi - S285363
     - Nobili Luca - S331461
   - Multilevel Queue Scheduler implementation:
     - Sannia Gabriele - S331385
     - Nobili Luca - S331461
4. Evaluate and benchmark the performance improvement achieved by the newly implemented solution:
   - EDF Scheduler Demo creation:
     - Sannia Gabriele - S331385
     - Pittalis Domenico - S283602
     - Nobili Luca - S331461
   - Multilevel Queue Scheduler Demo creation:
     - Nobili Luca - S331461