MÄLARDALEN UNIVERSITY
SWEDEN

ADVANCED MASTER THESIS ON COMPUTER SCIENCE:

ADAPTING MODE SWITCHES INTO THE HIERARCHICAL SCHEDULING

DANIEL SANCHEZ VILLALBA

SUPERVISOR: RAFIA INAM

EXAMINER: MIKAEL SJÖDIN

0. Abstract

Mode switches are used to partition the system's behavior into different modes to reduce the complexity of large embedded systems. Such systems operate in multiple modes in which each one corresponds to a specific application scenario; these are called Multi-Mode Systems (MMS). A different piece of software is normally executed for each mode. At any given time, the system can be in one of the predefined modes and then be switched to another as a result of a certain condition. A mode switch mechanism (or mode change protocol) is used to shift the system from one mode to another at run-time.

In this thesis we have used a hierarchical scheduling framework to implement a multi-mode system called Multi-Mode Hierarchical Scheduling Framework (MMHSF). A two-level Hierarchical Scheduling Framework (HSF) has already been implemented in an open source real-time operating system, FreeRTOS, to support temporal isolation among real-time components. The main contribution of this thesis is the extension of the HSF featuring a multi-mode feature with an emphasis on making minimal changes in the underlying operating system (FreeRTOS) and its HSF implementation. Our implementation uses fixed-priority preemptive scheduling at both local and global scheduling levels and idling periodic servers. It also now supports different modes of the system which can be switched at run-time. Each subsystem and task exhibit different timing attributes according to mode, and upon a Mode Change Request (MCR) the task-set and timing interfaces of the entire system (including subsystems and tasks) undergo a change. A Mode Change Protocol specifies precisely how the system-mode will be changed. However, an application may not only need to change a mode but also a different mode change protocol semantic. For example, the mode change from normal to shutdown can allow all the tasks to be completed before the mode itself is changed, while changing a mode from normal to emergency may require aborting all tasks instantly. In our work, both the system mode and the mode change protocol can be changed at run-time. We have implemented three different mode change protocols to switch from one mode to another: the Suspend/resume protocol, the Abort protocol, and the Complete protocol. These protocols increase the flexibility of the system, allowing users to select the way they want to switch to a new mode.

The implementation of MMHSF is tested and evaluated on an AVR-based 32 bit board EVK1100 with an AVR32UC3A0512 micro-controller. We have tested the behavior of each system mode and for each mode change protocol. We also provide the results for the performance measures of all mode change protocols in the thesis.

Table of Contents

1. Introduction

The complexity and size of real-time embedded system software is increasing day by day. This type of software is usually required to provide a wide variety of application scenarios for the same system. The vast range and rapid evolution of these application scenarios not only increase the overall complexity of the real-time embedded systems, but also demand more precise coordination and management among the different system functions. Moreover, a dynamic change in the application scenarios is required that usually modifies the behavior and services demanded by the user at runtime. All of these challenges together require a methodology that can handle the complexity of the system and also provide users with good results - something that is difficult to develop without investing a great deal of time and resources.

One way to avoid such a costly development is by simplifying the system. This is not done by restricting services, but by dividing the system into different parts whose development and maintenance become more manageable. Once this is done, they can be combined together once more to form the complete system. This is called Hierarchical Scheduling [10], dividing the system into a number of subsystems, each performing a specific application. An implementation of the *Hierarchical Scheduling Framework (HSF)* based on an open-source real-time operating system called FreeRTOS has been developed at MRTC [3, 4]. However, it does not solve the problem of runtime changes in the application scenarios.

The aim of this project is to adapt the existing HSF implementation with the dynamic changes in the application scenarios, hence developing a *Multi-Mode Hierarchical Scheduling Framework (MMHSF).*

1.1 Real-Time System

A real-time system is one that is restricted to timing constraints, also called “real-time constraints” [18]. This means that all functions must provide results within certain time limits.

Example: An Airbag system.

A car Airbag system is a classic example of a real-time system where timing constraints play a vital role. If a car has an accident, the airbag system must ensure the occupant’s safety; because if they are not inflated almost instantly a life could be lost. The specific response time for an airbag system is fixed at 1 ms (millisecond), so the embedded system responsible for the airbag deployment must take less than 1 ms to respond. To this end, a real-time system is used as it guarantees that the response time (the time from when the car receives information that it has had an accident to when the airbag deploys) is less than 1 ms. This time limitation is called “time constraint” and it must be established for every task in a real time device.

In summary, Real-Time systems are those systems that guarantee the performance of tasks within a specified time period. This feature makes real-time systems very accurate time devices, often used to accomplish critical tasks that should not exceed a certain time limit. This means that a delay in the task's execution could cause severe damage or failure (e.g. airbag system or a car's ABS). Real-Time systems are also used in high performance applications, where the quality of service depends on the response time of the system (such as video-conferences or Hi-Fi audio systems).

1.2 Multi-Mode System and Mode switches

Systems are typically *uni-modal* in nature, i.e.: they have only one mode to execute their tasks [20]. However, in a dynamic environment each task has to adapt its behavior according to different external or internal conditions. For example, consider a device powered by a limited battery resource, charged under normal conditions and behaving as a uni-modal system. At some point when the battery loses power, the device that is running must manage itself by reducing power consumption; for instance by reducing screen brightness or the processor load, etc. Each of these services needs to recognize the battery level and adapt by modifying its behavior accordingly. Moreover, in the example provided, there must be a *battery module* responsible for keeping track of battery power level, as well as other modules in charge of other features like screen or processor management. The latter must request data from the battery module in order to know the actual level of charge remaining in the battery; the system could be in *normal mode* when the battery is full, and could be in the *low battery mode* at other times. This example indicates the need to change the system's mode dynamically depending on the battery status.

A system that operates in different modes, where each has a particular functionality and a different timing behavior, is called a *Multi-Mode System [6]*. The system recognizes the conditions and switches from one mode to another at runtime. The system's tasks modify their own functionality and timing behavior when this occurs. This type of mode switch is controlled by a *Mode Switch Mechanism (or Mode Change Protocol)* [6].

Returning now to the example previously provided, a device behaves as a multi-modal system. When the battery drops below a certain threshold, the battery module recognizes this and notifies the system. The screen will now notice that the battery is low and provide a signal/message to the system. In turn, the system will switch modes, for instance, from "normal mode" to "low battery mode". This mode switch will make the services and modules modify their behavior, in some cases even canceling some old tasks or executing some new tasks.

In this project the main goal is to adapt the existing HSF implementation from a uni-modal system to multi-mode. However, this process is more complicate than it seems. The above example is simplified to facilitate comprehension of multi-mode systems, but there are many questions left unanswered: How quickly should the system switch to the new mode? How is the new mode communicated to the tasks? And what would happen if some tasks have nothing to do with the new mode? All of these questions have been investigated going back a very long time and multi-mode is, at present, a well-known technique used in embedded systems. On the other hand, these same questions have not been researched and applied to the implementation of simple hierarchical systems (HSF).

1.3 Related work

No work has been done in the literature with respect to the implementation of multi-mode hierarchical systems. A multi-mode schedulability analysis is presented in [11][12] and [13], and another analysis of a compositional system is found in [2]. The latter presents a multi-mode model and several techniques for analyzing systems that contain various applications. It also presents a case study about an adaptive streaming system that obtains better results with the multi-modal analysis than with the uni-modal analysis. There is a model for Mode-change Request that supplied numerous ideas to develop the MMHSF.

Some studies about multi-mode frameworks are presented in [14] and [15], where methodologies focusing on design reconfigurable, critical and complex embedded systems are presented. There are some other papers that deal with programming languages which support multi-mode, namely [16],[17], and [18].

A detailed Mode Switch Logic (MSL) algorithm is presented in [7]. This MSL implements coordination and synchronization of mode switch in component-based systems. This logic is implemented under the assumption that all of the components support the same modes, but a way to confirm this assumption is also proposed. A theoretical work that approaches the issue of multi-mode systems in component-based systems is explained in [1] and gives some algorithms that develop the ideas of the MSL presented in [7].

Finally a generic framework to implement a Multi-Mode Hierarchical system has been presented in [5]. It is based on a two-level HSF implementation in FreeRTOS and provides a framework for changing the system from uni-modal to multi-mode. It proposes the initial design details for the MMHSF implementation with the aim of making as few modifications as possible to the existing kernel, i.e.; the FreeRTOS, also used in [3] and [4] to develop the HSF implementation. Our work is the extension of that generic framework. We first implement a mode switch system to change the system's mode dynamically. We then present three different mode-switch protocols to change the system mode and their implementation details.

2. Background

This chapter provides the background behind the technologies which our work is based on. The first section explains real-time systems and deals with the features of a real-time operating system (RTOS). The second section gives a general overview of a specific real-time operating system, FreeRTOS, in which our implementation is based. The chapter finishes with a brief explanation about the Hierarchical Scheduling Framework.

2.1 Real-Time System and Real-time Operating System

For those not involved with the electronics or computer science fields, a task is defined as a set of instructions, data, and control information capable of being executed by the central processing unit in order to accomplish a certain objective [21]. As previously discussed, a real-time system ensures that its tasks will be executed within their time constraints. This feature is controlled by the operating system that governs the framework, which is called the real-time operating systems (RTOS). The RTOS is responsible for guaranteeing the execution of all tasks in a timely manner. To accomplish this goal, there are some features that allow the RTOS to meet time-constraints:

- An RTOS must be completely aware of the time outside the system (meaning the "real time"). To this end, it has been told that 1 ms in the system must be a real millisecond.
- It must rapidly switch from one task to another, spending as little time as possible in the task context-switch.
- The system must have some sort of interrupt subroutines, giving control of the execution to the scheduler as soon as possible.

All of these features are oriented to make the system quick and predictable in its responses. This is the responsibility of the scheduler. The scheduler chooses which task will be executed, when, and for how long. Generally, RTOS schedulers have two main policies:

- Preemptive Priority (also known as priority scheduler): it executes the highest priority task until the task ends or an event from a higher priority task needs to be attended. These priorities could be fixed or variable.
- Round Robin: the time is split into pieces or time slices and the scheduler executes tasks according to these time slices one after the other.

Both strategies need a suitable algorithm to be executed. It must be a deterministic algorithm, meaning that for a given input, it will always behave in the same way. The more deterministic the algorithm is, the more predictable the system will be. But sometimes this is not enough due to executed tasks which are often non-deterministic. This leads to one of the problems of RTOS: jitter. Jitter can be explained as the deviation between the executing time elapsed and the ideal executing time. The jitter phenomenon is well known, and schedulers keep must take it into consideration. But sometimes a task needs more time to be executed and as a result is not possible; this task would not accomplish its deadline. Such instances may cause different effects depending on the type of deadline:

- Hard deadline: if the task is not executed in time then it leads to a total system failure.
- Soft deadline: the task misses its deadline, however, the result of the executed task is

still valid even though it is not as good as it would have been had it been computed within its deadline.

We consider periodic execution of tasks in our system. This can be done in two ways:

- The task is programmed in a linear way, i.e.; the task starts its execution, executes its algorithm and dies. Here the RTOS is responsible for calling the function when its period is reached.
- The task is programmed in a circular way, i.e.; the task starts its execution and enters in a loop (usually an endless loop), executes its algorithm and waits until the next period. The task does this by calling a *wait* statement, which means that the task is already done and can be interrupted (preempted).

In the first method, it is the scheduler that has to keep track of the time to activate the task again and there is no need for additional structures to save the task status. In the second method the scheduler does not keep track of anything; instead it requires that the state of the task is saved somewhere (usually status registers) so that it may be restored when necessary. We use the second approach in our work.

2.2 FreeRTOS

FreeRTOS is an open source real-time operating system [8]. It is developed by "Real Time Engineers Ltd." mainly in C language and supports 31 different hardware architectures. It is very easy to use and modify. Its scheduler runs at the rate of one tick per milli-second by default, but it can easily be changed to any other value by setting the configTICK_RATE_HZ value in the FreeRTOSConfig.h file.

The FreeRTOS scheduler follows the fixed priority preemptive scheduling policy: execute the highest priority task until it is finished. Tasks with the same priority are scheduled using the round-robin policy. These tasks are in the form of an endless loop, calling a wait statement when they finish execution. At this moment the system saves the current state of the task in a structure called task control block *tskTCB*. It contains all the necessary information about the task's status. There is one of these structures per task, but they have to be stored somewhere. Since the system follows *fixed priority preemptive scheduling*, the task will be executed in a priority order. Therefore, the best method to save them is in a sorted queue. In fact there are two queues that manage this:

1. One queue is the *ready queue*, where the tasks are placed when they are ready to be executed. The ready queue is an array of *xList* elements that behave as an ordered queue, sorted according to task priority.
2. The second queue is the *release queue,* where the tasks go when they have been executed (when they are preempted). It consists of *xList* elements that sort the tasks by their next wake up time. This time tells the system when the task will be activated again.

It may happen that all tasks have been executed and there is no task in the ready queue, then the system will execute a special task called *idleTask*. This task is automatically generated by the operating system; it cannot be modified by the user, has the lowest priority, and never calls a *wait* function.

The system has a hardware timer that continuously counts the time. Every millisecond (ms) the system tick increments its time, storing the current time in a field called *xTickCount*. At each

system tick, the scheduler checks the release queue and checks the first task. If its wait time has expired then it moves the task from the release to the ready queue and checks the second task; if not, then it continues its normal execution. When a task is moved to the ready queue, it is compared to the task that is currently being executed (the current task stored in field *pxCurrentTCB*). If the new task has higher priority than that currently being executed, then a switch context is made (the current task stops its execution and saves its current state into its *tskTCB* field, then *pxCurrentTCB* is directed to the *tskTCB* field of the new task and the system restores the last state of the task stored in pxCurrentTCB).

In order to make the FreeRTOS run it is necessary to modify the *main.c* file, thereby creating all the tasks, declaring these tasks in the *main* function, and calling the *vTaskStartScheduler*. The *vTaskStartScheduler* function starts the scheduler and never returns. It starts the hardware timer, initializes registers, creates the idle task, and calls the scheduler.

2.3 Hierarchical Scheduling Framework and its implementation on FreeRTOS

The behavior described in the last section corresponds to the normal behavior of the FreeRTOS. The HSF implementation [3] is based on the FreeRTOS, hence special efforts are made to keep the HSF implementation compliant to the FreeRTOS. The HSF is composed of multiple subsystems (also called *servers*), each of which manages several tasks. These servers are scheduled by a global-level scheduler that governs the whole system. Each of the subsystems has its own ready and release-queue independent of other subsystems. These subsystems (servers) are like the applications in FreeRTOS by itself.

The servers have a set of parameters: *priority, period,* and *budget*. The priority has the same usage as in the task - to sort the servers so as to know the order of execution. The *period* indicates how often the server has to access the CPU for execution. And, the *budget* means the time the server has for execution in each period. When the server is activated (at every period) a variable called *remainingBudget* is set to the budget value, and at every system tick the executing server's *remaining-budget* is decreased by one. Once its value reaches zero, its budget expires; the server will be preempted and waits until its next period to be activated again. In our system we are using an *idling periodic* server type, whose execution process is explained below.

Example: Idling Periodic server Execution

Consider two servers, *S0* and *S1*, as Figure 1 illustrates. The *S0* has higher priority than the *S1*, and both have different periods, *T0* and *T1,* respectively. The arrival time for both servers is represented by an up arrow. *S0* has a smaller period than *S1*, and also a smaller budget (the budget is represented by the arrow's height). As can be seen in Figure 1, at the beginning both servers want to execute as their respective remaining budgets are at more than zero. However, since they cannot be executed at the same time, the highest priority server *S0* is executed first. The blue line represents the server execution. As time passes, the remaining budget decreases and eventually reaches 0. At this point in time, all tasks in *S0* are preempted and a context switch is made by the system, changing server from *S0* to *S1*. Now *S1* starts execution and its remaining budget starts to decrease. At time T0 the server *S0* will be activated again because its period has expired, returning the remaining budget value from 0 to the budget value. Since *S0* has higher priority than *S1,* it causes another context switch, from *S1* to *S0*. It is worth noting that *S1* was interrupted in the middle of execution, and its remaining budget is not 0. When the *S0* budget expires, it will be preempted and *S1* will start its execution again from the exact moment when it was previously interrupted. *S1* will finish its execution when its remaining budget expires.

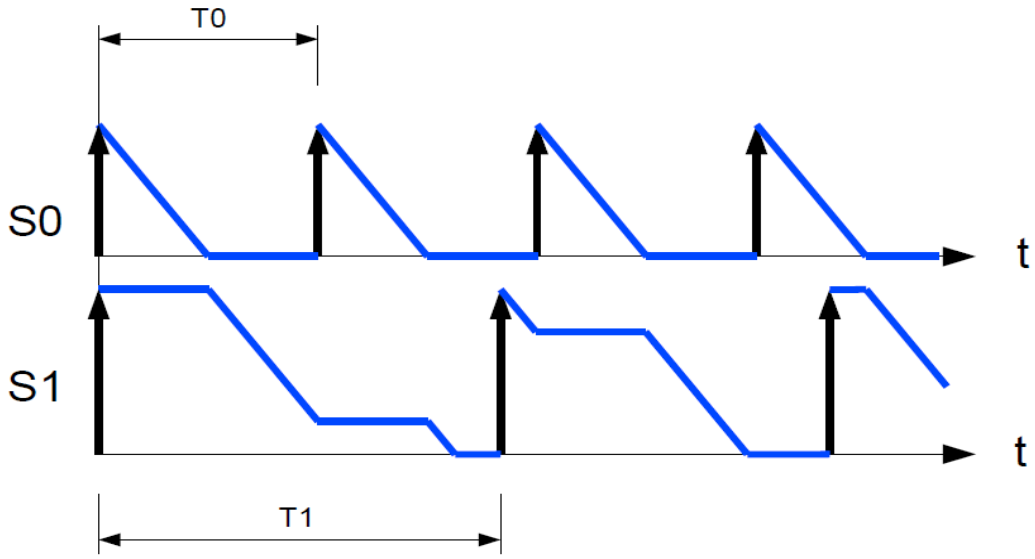

Figure 1: Servers execution in time.

There is a specific moment in Figure 1 when the remaining budgets of both servers are equal to 0. What is happening in the system? Neither *S0* nor *S1* are executing, so, what is the system executing? In this case, when all servers' remaining-budgets are equal to 0 then the system will execute the *idle server*. The *idle server* is a special server that is automatically generated by the system at the start of execution, when the function v*TaskStarScheduler()* is called. This server has the lowest priority, i.e.; 0, and infinite period and budget. Therefore, it will execute forever and it will never go to the servers release queue if no other high priority server is available in the system. Also, the *idle server* has priority 0 which means that whenever there is any other server, it will preempt the *idle server* and will be executed before the *idle server*. Inside this server there is only one task, the idle task of the server (as other servers have). There is no way that a user can create a new task inside this sever. Its function is to keep the system running when other servers have expired their remaining-budgets.

3. System Design

In this section we explain the system design. Our system design is an extension of the HSF implementation of FreeRTOS.

3.1 Assumptions

The assumptions are a series of barriers to limit the scale of the design, just to be clear what it should be performed. Later, some of these assumptions could be relaxed or changed for other less restrictive to allow the design grow further.

The basic assumptions are:

I) Fixed number of modes at the beginning of the execution. The user cannot declare new modes during run-time.

II) No shared resources between subsystems and modes. This assumption will facilitate implementation because no resource synchronization mechanism will be needed to manage the different resources the subsystem can share.

III) Fixed priority preemptive scheduling at both (global and local) levels. The behavior of the scheduler does not vary from one mode to another; it always works in the same way.

IV) Same task behavior. The task behavior (functionality and timing properties) remains the same in all modes; it can only select whether to execute or not (active or inactive task).

V) Only during the transition state, the local and global mode of the system may not be the same. The system mode will be changed when the entire subsystem's mode has changed to the new mode.

VI) Fixed number of servers. The number of servers does not vary from one mode to another. We assume that **all servers are active in all modes**.

Once the assumptions are defined, now it is time to describe a system model.

3.2 System model

A Multi-Mode Hierarchical Scheduling Framework (MMHSF) consists of different modes in a hierarchical system. The system can shift from one mode to another during the runtime. The proposed design of MMHSH is shown in Figure 2.

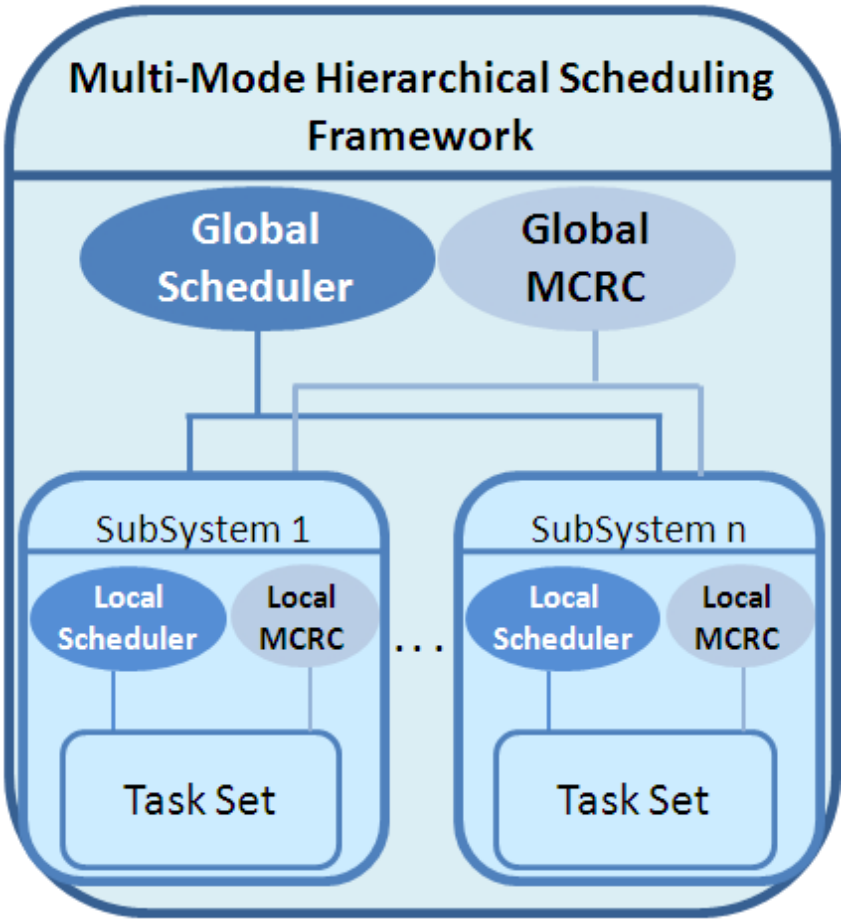

Figure 2: Multi-Mode Hierarchical Scheduling Framework (MMHSF) design details

In Figure 2 it can be seen that the system is modeled as a composition of various servers (or subsystems), and the *global scheduler* schedules which server has to be executed in which order (as in HSF). In this way the CPU time is divided among different servers. The *local schedulers* within each server then schedule their tasks according to their allocated timing resources (period, budget).

Furthermore, the system has several modes that determine the behavior of the subsystems and tasks, and it is able to switch from one mode to another. These changes are managed by the *Mode Change Request Controller (MCRC)*, which is responsible for capturing a request to change the mode (made by a task) and communicating it to another MCRC in the system. This mechanism is performed in a hierarchical manner, i.e.; a *global MCRC* receives a *Mode Change Request (MCR)* from a task within the server. The *local MCRC* transmits this request to the *global MCRC*, which, in turn, notifies the other local MCRCs to change the mode of the servers. This new mode indicates the current context of servers and tasks. As has been seen in the hierarchical scheduling framework section, each server has its own associated timing

parameters called *timing interface* (period, budget, and priority). In the multi-mode system these timing interfaces are defined for each mode separately, and it is possible for them to be different from one mode to another. The same thing happens to the tasks; they can have different timing properties in each mode.

The paragraph above briefly explained the behavior of the system when changing the mode - by switching the local modes of every server. Thus, it can ascertained that every server will have as many modes as the whole system. Based on *Assumption* V, these modes must be the same as the global mode, except in the transition state, where it is possible for them to differ.

To switch from one mode to another a task must trigger a *Mode Change Request (MCR)*. MCR is the mechanism to change the system's mode. The MCR is a request that is made by a task to the local MCRC in the server and then the demand is forwarded to the global MCRC. This request must specify (1) the target mode (or new mode of the system), (2) the mode change protocol that will manage the transition and, sometimes, (3) a deadline by which to perform the mode change. The server that triggers the request must behave according to the protocol specified by the *trigger function*. The *transition state* is the period during which the system is changing from the old mode to the new mode. A schema of the system during the transition state is shown in Figure 3.

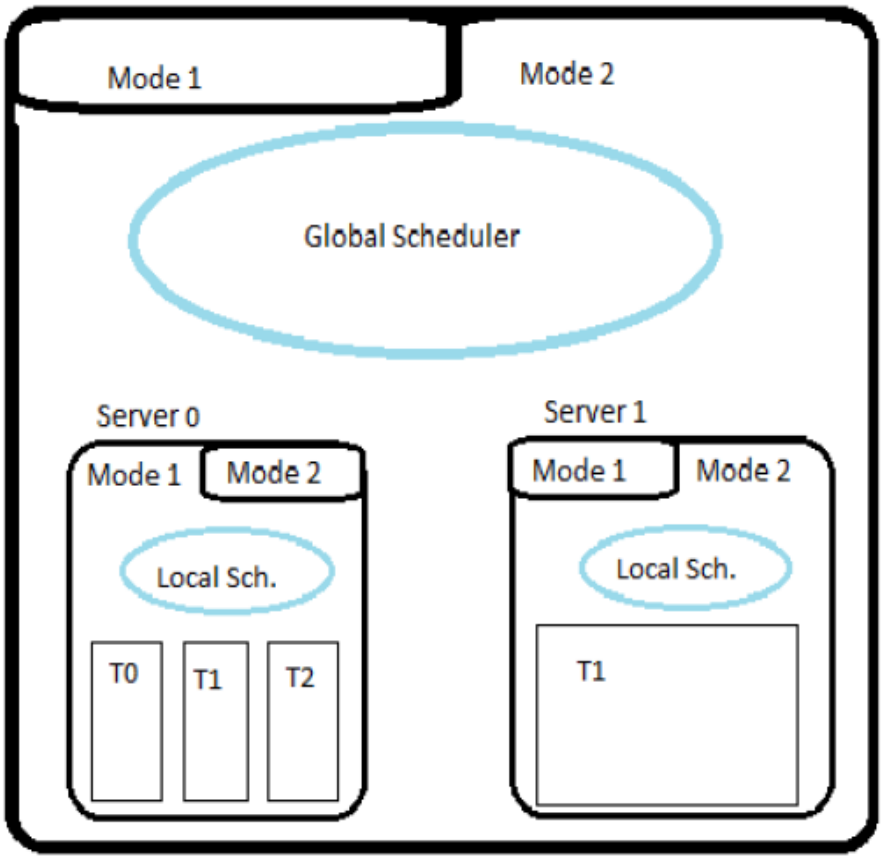

Figure 3: System schema during transition state

The task *To* of *Servero* triggers the MCR. The local MCRC instantaneously forwards the request to the global MCRC. The global MCRC then communicates this request to the local MCRCs in other servers (*Server1* in Figure 3*)* to automatically change their modes (to *Mode2*). At this point the whole system is in *Mode2*, except *Server1,* which is still in the previous mode, *Mode1*.

At this point we could have two different scenarios (according to *Assumption IV*): the triggering task is active in the new mode, or the triggering task is inactive in the new-mode. The first scenario is “simple” to solve: the triggering task will continue executing according to the **fixed priority scheduling**, i.e.; if the task has the higher priority in the new-mode it will continue its execution, otherwise it has to wait in the ready queue. The second scenario is more complex and requires some external help to be solved. At this stage the other parameters of the MCR come into play, namely protocol, and the deadline, which will be explained in the next section.

3.3 Mode change protocols

An interesting question was put forth in the previous section: What happens to the task that triggered the MCR? While a brief answer was given, a more extensive, detailed explanation now follows.

Focusing on the first scenario described (the task is active in both modes), there are two possibilities. On the one hand, if the task has the highest priority in the new-mode, then the system will continue executing the task. On the other hand, if the task does not have the highest priority, then the system will suspend the task (as if the task has reached a *wait* statement) and will add it to the ready-queue based on its priority.

In the case of the second scenario (the task is active in the old-mode but inactive in the new-mode), what happens to the task? To answer this question we have defined a set of mode change protocols; they are the **complete-protocol**, the **abort-protocol**, and the **suspend/resume-protocol**. They are explained as follows:

- **Complete-protocol:** the server will finish all tasks before it switches the system to the new mode. In this protocol, we use a deadline that defines a time limit to complete the task. If the task takes more in its completion than the defined time limit, then the system will force the mode-switch to the new mode, acting like the suspend-resume protocol.
- **Abort-protocol:** Using this protocol, the system stops executing all tasks immediately and changes the mode as soon as possible. If a task is inactive in the new mode, then it releases the possible shared resources it had locked up. When the system returns to the old-mode again, all tasks are activated from the start.
- **Suspend/resume-protocol:** Using this protocol, the system suspends all tasks in the old-mode, switches to the new-mode and, when the system returns to the old-mode again, it resumes those tasks from the point where they had been previously suspended.

The two first protocols, Complete and Abort, are mostly clear in their functionality: allow all tasks to finish until till their end or stop the execution of all tasks at at the same time, respectively. However, in the case of Suspend/resume-protocol there are some questions that do not have such clear answers and it is worth discussing them.

I) When an MCR is triggered, what happens with the remaining budget of the servers?

As explained in section 2.3 on hierarchical scheduling, each server has a remaining budget which is equal to the servers' capacity/budget at the start of the server execution and decreases when the server executes. In the multi-mode context, each server has a different budget and remaining budget for every mode. For complete- and suspend/resume-protocols, when an MCR is triggered, the system saves the server's remaining-budget from the old mode and restores the remaining-budget of the new mode of every server.

If the protocol selected for the MCR is the abort-protocol, all servers and tasks will start their execution from time zero, meaning the system does not store the remaining-budget. The server's remaining-budget for the new mode will be set to the budget value for the servers in the new mode.

II) **Suppose there is an initial MCR from *Modeo* with suspend/resume protocol specified, and a second MCR with abort protocol to *Modeo*, what would happen?**

Using *suspend/resume protocol* the system suspends all tasks and servers of the old-mode in the system and then resumes all tasks and servers of the new-mode. When the tasks and servers are suspended, their status for old-mode is stored and when this old-mode is returned to again, regardless of what the second MCR protocol is, it will resume the task and the servers will move everything to the ready list (ready task list for tasks, and ready server list for servers)

III) What would happen to the task in the release queue at an MCR request? When would the task be activated?

It is well know that tasks are self-triggered, i.e.; each task indicates to the scheduler when it wants to be "activated" again by means of a time-based *wait* statement. In the HSF implementation, this statement makes the system move the task to the *release queue* (from the ready queue) and when the specified time has elapsed, it then moves the task back to the *ready queue*.

In the MMHSF system, it is possible for an MCR to be triggered while a task is waiting in the release queue. If this task is inactive in the new mode, the system will still keep track of the actual time the task was waiting until the MCR was made. It will then compute the remaining time the task has to wait and save it in a data structure (a new field that stores this time value for every task in every mode). When a new MCR is triggered to switch the system to the old-mode, then the system will recover the remaining time for this task in the current mode and, based on the current time, computes when the task has to be activated again (to move it into the *ready queue)*.

Figure 4 illustrates how this activation is made. In mode M0, when the first MCR is made, task T0 should be activated after 2us. This time is stored in the system when the mode is changed to M1. Later, at the second MCR, when the system changes its mode back to M0, the task T0 is activated after 2us.

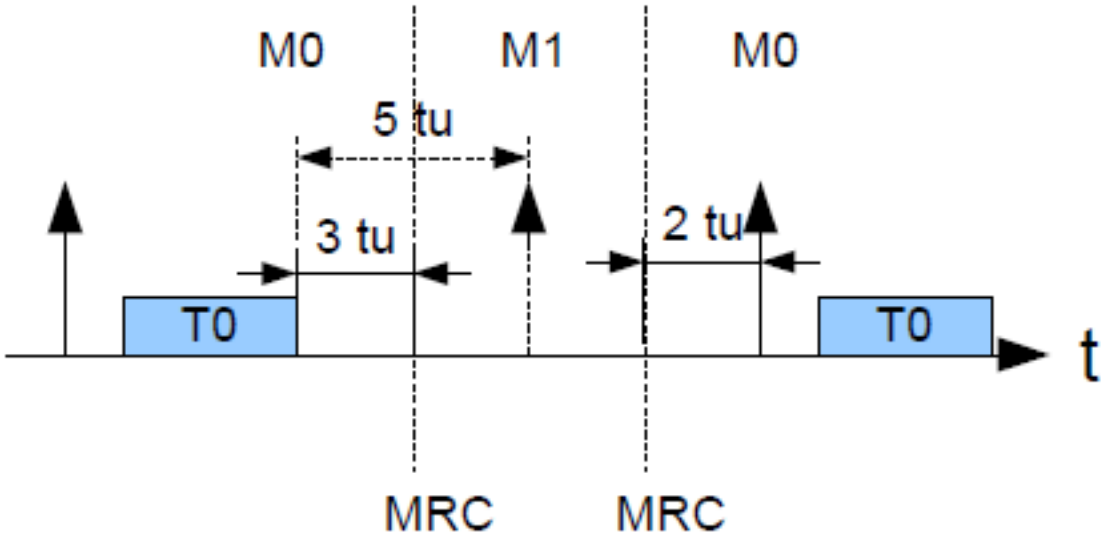

Figure 4: re-activated task in the suspend/resume context

This method is called **frozen-time** and functions as follows:

When a *wait* statement is called by a task, the system computes the next activation time of

the task (saved in the field *xReadyTime*, in Figure 5 it is represented by the arrow called "t'"). When an MCR is triggered, the system obtains the current time *(* "t1" in the diagram*)* and saves it in a structure that saves the time when an MCR is executed by the system (called *xModeTickCount*).

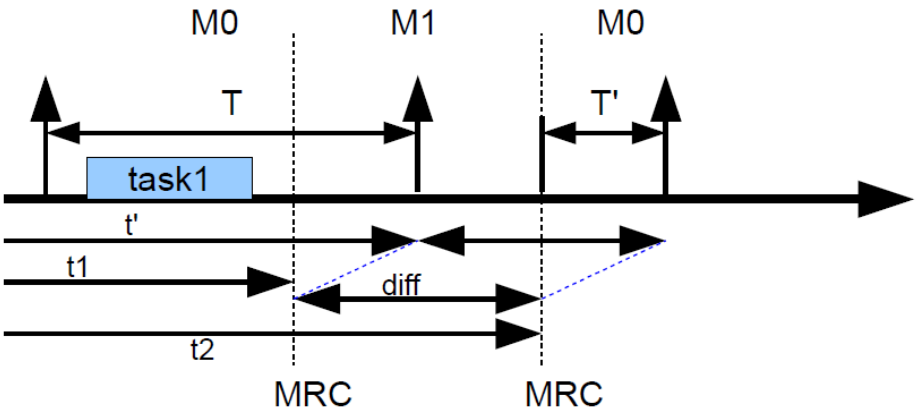

Figure 5: Frozen-time procedure.

When a new MCR is triggered to restore the system to the old mode (represented as arrow "t2" in the diagram), the system must compute how long the task must remain asleep. For this purpose it is necessary to know when the MCRs have been executed. Using the field *xTickCount* the system can compute how long the task was inactive by doing this operation:

$$diff = t2 - t1$$

Then, when the task is moving into the release queue during the switch mode, the next awake time of the task is updated by adding the "diff" value:

$$t'' = t' + diff$$

where t' is the last activation time and t" is the next activation time. This value must be stored in *xReadyTime* and in the *xGenericItem* value, and simply remain to later move the task into the *pxDelayedTaskQueue*.

IV) What would happen with the budget and the period in the suspend/resume mode-change context?

If we consider the Figure 6 context, in which server S1 (with the highest priority) has spent all its budget and server S2 (lowest priority) makes an MCR during its execution. The remaining budget of S2 will be saved in the *timing interface* field mentioned above (the same is done for S1, but the remaining budget is 0 in this case, so it is not worth analyzing), as well as the current time in which the request was made in order to keep track of the spend time according to the server's period. Then, the system enters in a new-mode, M1. While the system is in the new mode, the periods for both servers are over, and they need to be "activated" again. However, in this scenario both servers are inactive in M1, so they continue waiting without being executed. After some time another MCR is made to change the system to mode M0. At this point the system encounters four different types of servers: those which were active in M1 and remain active in M0; those which were inactive in M1 and remain inactive in M0; those which were active in mode M1 and are inactive in M0; and those which were inactive in M1 and are active in M0. Nothing can be done with the first and second type, and the third was already explained in the first part of this example.

The interesting procedure here is for the fourth type. We can split these servers into two subtypes: firstly, the server that was being executing when the first MCR was made, i.e.; it was in

the *ready queue;* secondly, the server that was waiting when the first MCR arrived, i.e.; they were in the *release queue*. For the first type the procedure is simple: simply restore the old *remaining-budget* (which was previously stored, in the first MCR) and move the tasks to the current *ready-queue*. The procedure for the second type is more complex: the system has to compute how long they need to wait in order to the period constraints, and it saves this time in the *xReadyTime* field. Finally, the system has to move the servers to the *release-queue*. However, since assumption VI requires all the servers to be active in all modes, this procedure is not employed or implemented (but the code is already prepared to support this feature in future).

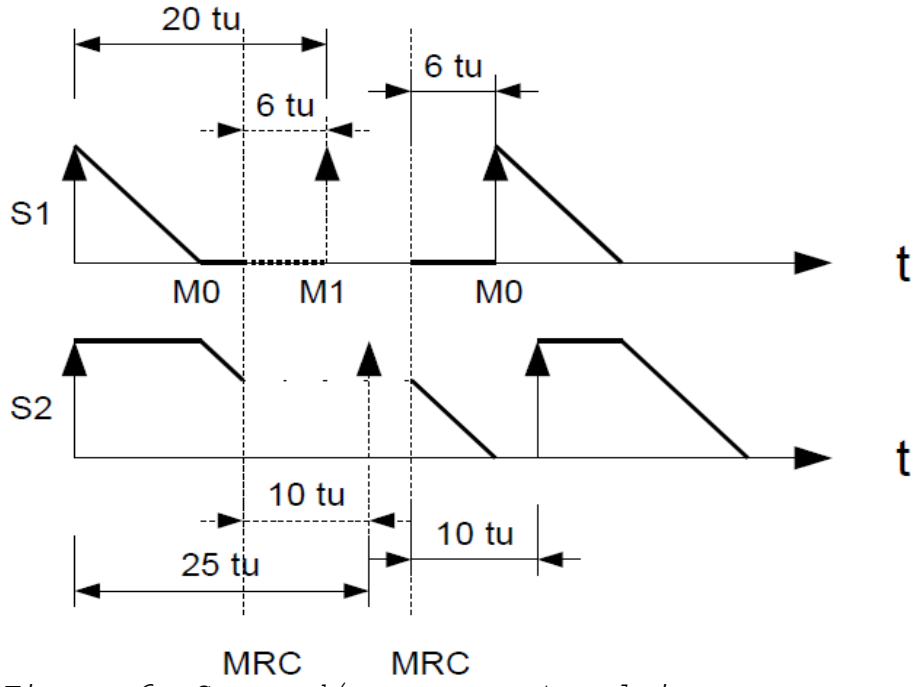

Figure 6: Suspend/resume protocol in servers.

This procedure is also **frozen-time** and it functions the way that **frozen-time** does for tasks. Since servers are always active (Assumption VI) there is no need to consider what would happen to a server.

4. Implementation

In this section we describe the implementation details, which include data structures, new and modified API, and new and modified macros of our code.

4.1 Data structures

In order to achieve the design proposed it is necessary to modify and add some new data structures to the existing HSF implementation. The modifications are discussed as follows:

- **Tasks Ready queue**: the two-dimension queue is now substituted by a three-dimension structure with the following form: readyTask List [x Number of modes] [x priorities][tasks of priority x], a separate two-dimension queue for modes 0 to n-1 is shown in Figure 7.
- **Tasks Release and Overflow queue**: now it is a two-dimensional queue, one separate queue for each mode, as shown in Figure 7.
- **Task Control Block tskTCB**: The TCB structure also adds three more fields: one that determines if the task is active or inactive in every mode (*xTaskBehaviorMatrix*), another one that specifies if the task is suspended or not(*uxIsSuspendedFlag*), and a final field that provides the last mode in which the task was active(*sLastActiveMode*), as shown in Figure 7. Furthermore, the task's priority is substituted by an array, one priority per mode.
- **Server Parameter List**: The budget, priority, period and remaining-budget are clustered in a unique structure. There is a separate array of this structure called *xServerParameterList* for each mode in the system (see Figure 7).

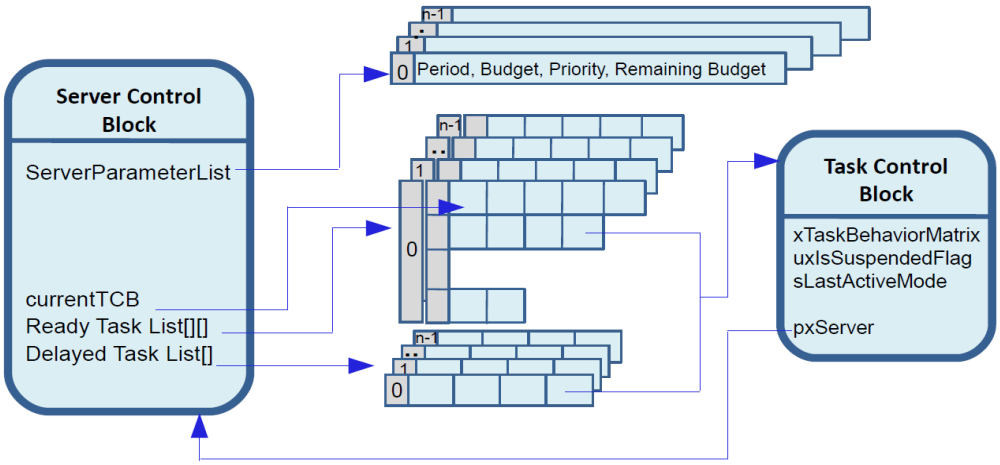

Figure 7: SubSCB and TCB modified

Some new variables and structures are required to make the system work properly; they are shown in Figure 8.

- **Server Ready queue, server Release and Overflow queue**: now each is a two-dimensional queue, one separate queue per mode as shown in Figure 8.
- **Server Control Block SubSCB**: It adds a field that determines the current mode in which the server is executing (*sLocalCurrentMode*). Also, it contains two flags to indicate where the server is: in the ready, release, or overflow queue

(*uxInReadyQueueFlag* and *uxInOverflowQueueFlag*), as shown in Figure 8.

- A variable that contains the system's current mode is (*sGlobalCurrentMode*).
- A variable that specifies the protocol which the system is going to follow during the mode switch (*sSwitchModeProtocol*)
- A structure that contains the times when every mode was switched off (*xModeTickCount*). These times, combined with the *tskTCB* field that provides information about the last mode in which the task was active, is very useful for computing how long the task has been inactive. This was addressed in the previous chapter when the explanation on **frozen-time** was set out.
- A flag to indicate if there is any mode-switch in execution following the complete protocol (*xCompleteFlag*).
- A variable that saves the new mode when a mode-switch is in execution following the complete protocol (*sIncompleteMode*).
- A variable that counts the time spent during the mode-switch under the complete protocol (*xCompleteDelayedTime*).
- A flag to indicate if a mode-switch is in execution or if the mode-switch cannot be done (*xSwitchInCourseFlag*).

In Figure 8 the changes performed in the HSF implementation can be observed, as well as how the servers' queues are now two-dimensional.

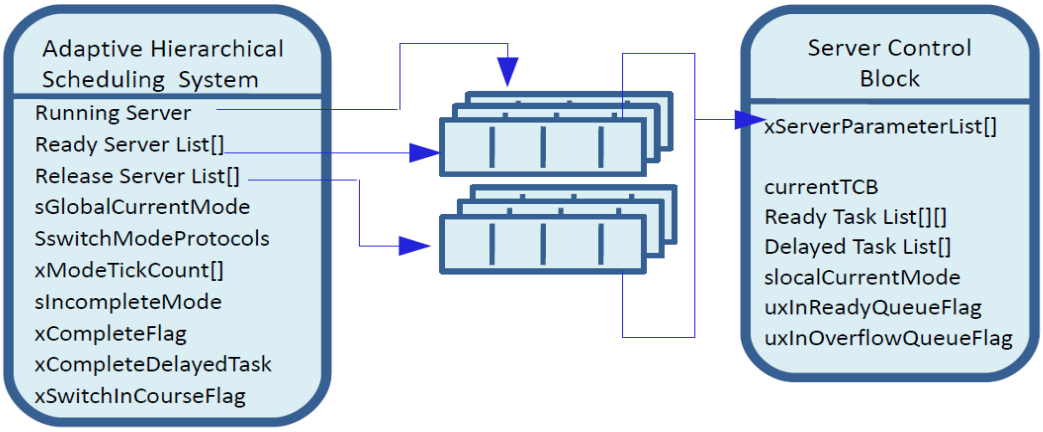

Figure 8: Additions to the HSF data structure

It is also necessary to declare a new structure that contains information about all servers and all tasks contained within them. It is necessary to declare this new structure because otherwise, when a mode-switch is performed, the server and task portability to the new mode requires the system to spend a lot of time looking for servers in the queues, but it spends even more time looking for all the tasks in the ready, release, and overflow queues. This structure is modeled as an array: one element per server with the following fields, as is shown in Figure 9.

- A pointer to the server that the element is represented (*pxServer*).
- An array that contains all tasks in this server, active or inactive (*pxTaskArray*).

Since we are assuming that the number of servers and tasks may vary during the execution, i.e.; new tasks and servers can be **created during run-time**, this structure must be dynamic and adaptable to the changes in both servers and tasks. In the case of tasks, they are allowed to be deleted so the structure must have a procedure for erasing a task from the tasks array. The

way to make an array dynamic is to declare a pointer of the element's type. For this purpose two new types are declared. The first is the *taskArrayElement* type, which is a pointer of type *tskTCB*. With it, a pointer to a *taskArrayElement* can be declared, which means a double pointer to a *tskTCB* structure, i.e.; a new array structure has been created, one where the elements are *tskTCB* pointers. In this new type all the tasks contained in a given server can be grouped together. Consequently, a second structure is required, one that contains the array described above, as well as the *subSCB* pointer of the server to which the tasks of the array belong to. That structure is called *serverArrayElement* and represents a server. Finally, a dynamic structure with all the servers and tasks must be created. For this purpose a global field is declared - *pxAllServersArray*: a pointer of type *serverArrayElement*. This pointer is a dynamic array that allows using functions *pvPortMalloc* and *pvPortRealloc* to dynamically allocate and deallocate servers in the system. The same functions are used to manage the *taskArrayElement* pointer that contains all the tasks in a concrete server.

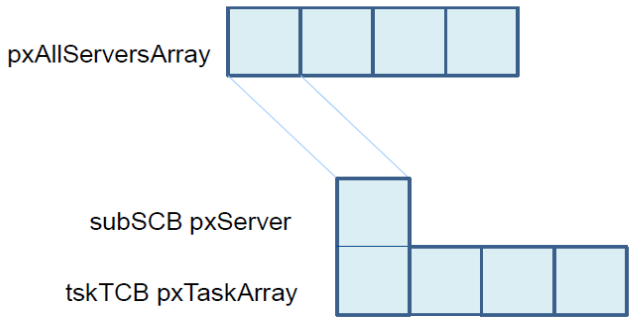

Figure 9: pxAllServersArray structure.

The total number of modes and different mode-change protocols are defined in the configuration file, providing the developer with the freedom to create new protocols.

With all these variables and data structures, now the system is capable of sustaining different modes within hierarchical scheduling. All that is needed now is the procedures/function to manage them properly. So as to correctly execute the new system, it is necessary to modify some functions and macros and to create new ones. In the next section all of these modified or newly created routines are explained.

We will continue by describing how the system was modified to support the multi-mode feature: the changes made to the functions and the newly created functions. Most of the changes made to the functions are based on the fact that we have redefined some data structures (not just the ready and release queues but also the priorities in both servers and tasks structures). Other changes are oriented towards easing the mode-switch mechanism or the performance of the whole system with different protocols behavior.

Most of the new functions are targeted in the mode-switch procedure. We have tried to keep the system's behavior compatible with the FreeRTOS code and its HSF implementation. The original system can be used by setting the *configMULTI_MODE value* to zero in the *FreeRTOSConfig.h* file. We use compiler directives such as *#if(configMULTI_MODE)*, *#else* and *#endif*. If *configMULTI_MODE* is set to 1, then the constant *N_MODES* must be set to higher than 1. *N_MODES* determines the total number of modes in the system.

Furthermore, there is another change in the behavior of the system that eases the mode-switch procedure. That change concerns the server's remaining-budgets: in the HSF implementation, when a server spends all its remaining-budget, the ready time is updated to the next period, the remaining-budget is restored and the server goes into the release queue.

When a server spends all its remaining-budget, the ready time is updated to the next period, the server is moved into the release-queue and the remaining-budget remains 0. When the server is preempted and moved once again to the ready-queue then the remaining-budget is set to the server budget. This allows the system to behave in an ideal way, and when an MCR is triggered the system can then pay attention solely to the server's remaining-budget so that it know where the server must be moved (to the ready or release queue).

4.2 Modified API and Macros

Here we present all the modified API and macros.

4.2.1 Modified Macros

Macros are sorted by order of appearance in the code. The macros and their descriptions are given below:

- `prvAddServerToReadyQueue( pxSCB )`

Inserting the server into the ready queue. Ready queue is a priority array sorted according to the priority of the server in a particular mode.

- `prvAddServerToReleaseQueue( pxSCB )`

Inserting the server into the release queue. Release queue is a priority array sorted according to the server's next activation time (`xReadyTime` of the server).

- `prvAddServerToOverflowReleaseQueue( pxSCB )`

Inserting the server into the overflow release queue. Overflow release queue is a priority array sorted according to the server's next activation time (*xReadyTime* of the server).

In all of these macros the value of the flags *uxInReadyQueueFlag* and *uxInReadyQueueFlag* is updated properly to the destination of the server.

- `prvAddTaskToReadyQueue( pxTCB )`

In this macro the task is inserted in the server ready queue and sorted according to its priority. The *uxTopReadyPriority* is also updated if is necessary. To make this function work properly it is necessary to correct the access to the new priority array and add the task to the proper queue, sorting it now according to both priority and mode.

- `prvChooseNextIdlingServer()`

This macro accesses the ready server list (called *readyServersQueue*) and selects the highest priority server as the next to be executed. For this purpose, we use the *readyServerQueue* array and access to the priority by means of the *xServerParameterList* array.

- `prvCheckDelayedTasks(pxServer)`

This macro looks through the release queue of the server and finds the task whose *xReadyTime* has expired and adds it to the ready queue. Since the *pxDelayedTaskList* is an array, this macro has been changed to ensure proper access to the queue.

4.2.2 Modified API

Functions are sorted by order of appearance in the code. As previously explained, most of the changes consist of updating the functions to the new form of the data structures, mainly for the queue arrays, the priority array in the *tskTCB* structures and the *xServerParameterList* in the *subSCB* structures. Most of the changes consist of the same function, but instead of a single variable assignation it has a *for loop* statement to perform this assignment for each mode.

A clear example of that is the *prvInitialiseServerTaskList*. In the old system it consisted of one *for* structure to initialize the ready *pxReadyTaskList* array, but since this structure is a two-dimensional array in the new code, it needs nested *for* structures. Similarly, other queues (Delayed and Overflow queues) also need a nested for statement instead of a single *for* structure.

- `void prvOverrunAdjustServerNextReadyTime( subSCB *Server)`

This function works when *configGlobal_SRP* is set to 1. The function computes, if the remaining-budget of a server is expired, what it will be the next time when the server will be preempted and added to the ready queue. Some changes are related to the ready and release queue and to the *xServerParameterList*. Furthermore, this function is responsible for setting the remaining-budget when the server goes to the ready queue. In order to make this change a line of code had to be reallocated. Said line is now allocated in the function *vTaskIncrementTick*, when the system is looking for servers to awake and move to the ready queue.

- `void prvAdjustServerNextReadyTime( subSCB *pxServer )`

This function is not used if the constant *configGLOBAL_SRP* is set to 1. Its functionality is to move *pxServer* to the proper list and update the server's next ready time. Since the system is built to support shared resources this function is not used. In any case it has been modified to properly work in the new system, also with the new remaining-budget behavior.

- `void prvInitialiseTCBVariables( tskTCB *pxTCB, const signed char * const pcName, unsigned portBASE_TYPE *uxPriority, const xMemoryRegion * const xRegions, unsigned short usStackDepth )`

This function initializes the TCB variables.

- `void prvInitialiseServerTaskLists( subSCB *pxServer )`

This function initializes the server's list contained within the SCB (see Figure 7): ready, delayed and overflow, and as they possess a new dimension they need to be initialized with a *for* loop.

- `void prvInitialiseGlobalLists(void)`

This function is responsible for initializing the *xReadyServersList*, p*xDelayedServersList* and *pxOverflowServersList* structures among others. Since they became an array of type *xList* they need to be initialized with a *for* loop.

- `signed portBASE_TYPE prxRegisterTasktoServer(tskTCB * pxNewTCB, subSCB`

```
*pxServer)
```

This function is responsible for associating the task in *pxNewTCB* to the server pointed by *pxServer*. Two modifications are made in this case. The first is related to the new structure created: *pxTaskArray*; this is where the new task is registered to the structure in the form of the new array's element. The second change concerns the behavior of the tasks. In the HSF tasks are included in the ready list. Now, in the MMHSF, the task may be inactive in the current mode, which means that the task could not be included in the ready or release lists. In this case the task is marked as “suspended” and not included in any list.

- `signed portBASE_TYPE prxServerInit(subSCB * pxNewSCB)`

This function is responsible for registering the server to the scheduler. In addition, this function is responsible for updating the *pxAllServersArray* structure, adding a new element to the array by means of function *pvPortRealloc*. As we have assumed that all servers are initially active in all modes, there is no need to select where to put the *subSCB*; it must go into the *xServerReadyList*. For the purposes of future extensions, choosing whether the server is active or inactive can be done in this function by asking structure *xServerBehaviorStructure* (already implemented but not used).

- `signed portBASE_TYPE xIdleServerCreate(void)`

This task was modified to properly set the idle server parameters using a *for* loop.

- `signed portBASE_TYPE xServerCreate(xServerParameters *pxServerPL, xServerHandle *pxCreatedServer,unsigned portBASE_TYPE *xServerBehaviorMatrix)`

This function is used to create the server structures. It has important modifications in the header, substituting all the server parameters (priority, period and budget) using a pointer to *xServerParameters*. This pointer contains an array of length *N_MODES*, which is assigned to the *xServerParameterList* field in the *subSCB* structure.

- `void prvScheduleServers(void)`

This function also has an important role in system behavior. Here the remaining-budget decreases at every system tick. When the remaining-budget reaches 0 or if another server with higher priority activates, then function *prvChooseNextIdlingServer* is called to select a new server to run, unless there is an uncompleted mode-switch with the complete protocol. This procedure is explained in detail in the next section: 4.2 Created Functions.

- `signed portBASE_TYPE xServerTaskGenericCreate(pdTASK_CODE pxTaskCode, const signed char * const pcName, unsigned short usStackDepth, void *pvParameters, unsigned portBASE_TYPE *uxPriority, xTaskHandle *pxCreatedTask, xServerHandle pxCreatedServer, portSTACK_TYPE *puxStackBuffer, const xMemoryRegion * const xRegions, unsigned portBASE_TYPE *xBehaviorMatrix)`

The basic purpose of this function is to create a new task in the server. Many changes have been made to this function as well. Firstly, parameter *uxPriority* is now a pointer to *portBASE_TYPE* and contains an array of *N_MODES* length. Secondly, it has a new parameter added at the end, a pointer to *portBASE_TYPE* that contains an array again of *N_MODES* length with the behavior that the task is going to follow. This behavior matrix is the same as that used in the *prxRegisterTasktoServer* function, and it determines the response of the task when

a MCR arrives. In the function body there is another modification related to the new way of accessing the priority.

- `portTASK_FUNCTION( prvServerIdleTask, pvParameters )`

The function can also be called an *idle* function. This function also has an important role in the mode-switch in the complete protocol. This task is executed in the system at two moments: when there is no other task to be executed in the server or during the time between a call to the *vTaskDelayUntil* function and a system tick. This function is used to finish a mode-switch that follows the complete protocol. In the next section (4.3 New API) this procedure is explained in detail.

- `signed portBASE_TYPE xTaskGenericCreate(pdTASK_CODE pxTaskCode, const signed char * const pcName, unsigned short usStackDepth, void *pvParameters, unsigned portBASE_TYPE *uxPriority, xTaskHandle *pxCreatedTask, portSTACK_TYPE *puxStackBuffer, const xMemoryRegion * const xRegions, unsigned portBASE_TYPE *xBehaviorMatrix)`

The parameters of this function were modified in the same way as the *xServerTaskGenericCreate*.

- `void vTaskDelete( xTaskHandle pxTaskToDelete )`

This function was modified to properly remove the element of the field *pxTaskArray* in the *pxAllServersArray* structure corresponding to *pxTaskToDelete*: firstly the array position is overwritten using a *for* loop. Then the field is dynamically re-sized using the *pvPortRealloc* function.

- `void vTaskDelayUntil( portTickType * const pxPreviousWakeTime, portTickType xTimeIncrement )`

This function was modified to properly add the task to the delayed or overflow queues, taking into consideration the server's mode.

- `void vTaskDelay( portTickType xTicksToDelay )`

This function was modified in the same way as the *vTaskDelayUntil* function.

- `unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask )`

This function was also modified to provide only the priority of the task in the current mode of the server.

- `void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE *uxNewPriority )`

As was done with all of the functions presented above, this function was also modified from the header to accept a pointer to *portBASE_TYPE* containing an array of *N_MODES* elements. In addition, this function features some modifications when it tries to determine if it could be the next current task.

- `void vTaskResume( xTaskHandle pxTaskToResume )`
- `portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume )`
- `signed portBASE_TYPE xTaskResumeAll( void )`

These functions are responsible for restoring the execution of the system when a suspend function has been called before. A modification arises in these three functions when they are comparing priorities, due to the data structure changes.

- `void prvSwitchServersOverflowDelayQueue(xList * pxServerList)`

This function has some modifications in the way to exchange queues, i.e.; now the queues are formed by arrays, then a *for* loop is needed.

- `void vTaskIncrementTick( void )`

This function is called in every interruption of the timer/counter and is responsible for increasing the time of the system. This function has several changes. The first is oriented towards ensuring a proper mode-switch when it is triggered from an interrupt subroutine by setting *xSwitchInCourseFlag* to 'true'. In this way, if an interruption occurs during the tick increment function, the MCR is ignored and does not interfere with the system behavior. The second change is due to the possibility of a counter overflow. As in the previous function the queues are now arrays, so the proper way to manage them is through the *for* loop, calling the *prvSwitchServerOverflowDelayedQueue* three times per mode, once per queue (ready, release and overflow queues). There are also some changes related to the way in which the servers queue is properly accessed. And, finally, there is another important change: when a server is added to the ready queue then the remaining-budget of the server is set to the budget value of the current mode of the server.

- `void vTaskSwitchContext( void )`

This function sets the pointer currentTCB to the TCB of the highest priority task that is ready to run. In this function two changes are made. The first is related to the way the *pxReadyTaskList* of the server is accessed, as it is now an array of queues. The second involves releasing the system grant permission to the MCR, which is blocked in the *vTaskIncrementTick*, once again allowing the mode-switch from an interrupt subroutine.

- `signed portBASE_TYPE xTaskRemoveFromEventList( const xList * const pxEventList )`

This function removes a task from both the specified event list and the list of blocked tasks and places it in a ready queue. This function has a modification during comparison of two priorities (the current task priority against the top task priority in the events list).

4.3 New API

In this section we discuss how the system changes among the different modes and all the newly created APIs to accomplish its purpose. The order of the functions is selected to ease the reader's understanding of different system behaviors. There are some references to functions explained in the last two sections (4.1 Data Structures and 4.2 Modified Task and Macros).

- `short prsReturnAllServersArrayIndex(subSCB *pxServer)`

This is an auxiliary function used to find the index that corresponds to the pointer *pxServer* inside the structure *pxAllServersArray*. The execution time of this function depends on the position of the server that is being searched for (the servers are ordered by creation time, the first created is the first array element). This function returns either the array's index for the server or -1 if the server does not exist.

- `short prsReturnTaskArrayIndex(tskTCB *pxTCB)`

This auxiliary function is used to find the index that corresponds to the pointer *pxTCB* inside the field *pxTaskArray*, among the structure *xAllServerArray*. This function uses the *prsReturnAllServersArrayIndex* to find the server to which the task belongs. Also, the time spent here to perform the search is variable and depends upon the position of the server in the structure and the position of the task in the array (the task has the same pattern as that of the servers). If the task is found, the function returns the index of the task inside the *pxTaskArray* field. If the task does not exist then it returns -1.

- `unsigned portBASE_TYPE xTaskChangeTaskModeBehavior(short mode,unsigned portBASE_TYPE xBehavior)`

As has been discussed, a task may be active or inactive in the different modes. This choice is saved in a field called *xBehaviorTaskMatrix* contained in the *tskTCB* structure of the task. This behavior matrix can be configured at the creation of the task. This function can also be used to modify the behavior of the current task in a concrete mode with the value of *xBehavior*. This function returns *pdFALSE* if *mode* is equal or higher than *N_MODES* value and if *mode* is equal to the current mode in the server, otherwise, it returns *pdTRUE*.

- `unsigned portBASE_TYPE xTaskChangeServerModeBehavior(short mode, unsigned portBASE_TYPE xBehavior)`

This function performs the same operation as *xTaskChangeTaskModeBehavior* but for the current server. The requirements for success are the same, but since the system assumes that all servers are active nothing can be done with the server's behavior matrix; therefore this function is simply created as a **guideline for future developers**.

- `void vTaskStartModeScheduler(short defaultMode)`

This function initializes all the variables and fields related to the mode-switch. It determines the initial system mode, sets the field *xModeTickCount* to zero, deactivates the flags *xCompleteFlag* and *xSwitchInCourseFlag*, initializes the *xCompleteDelayedTime* to zero and it gives *sSwitchModeProtocol* the default value of *SUSPEND_RESUME_PROTOCOL*. This function must be called before any other in the system as it determines the system's mode, and, above all, so much depends on this field, such as the server's initialization or the task

registration. In addition, this function must not be called twice in the same system execution.

- `void vTaskChangeProtocol(short sNewProtocol)`

This function is responsible for changing the protocol for the mode-switch. The different protocols are defined in the *FreeRTOSConfig.h* file and they are the same as those described in section 3.3 on mode change protocols. This function is not executed properly if *xSwitchCourseFlag* is set to 'true', leaving the function without changing the protocol.

- `short sTaskGetCurrentSystemMode(void)`

This function returns the system's current mode.

- `portBASE_TYPE xTaskIsCompleteInCourse(void)`

This function returns the value of *xCompleteFlag*, informing whether there is an unfinished mode-switch that follows the complete protocol. Due to the behavior of the system the only mode-switches that can be unfinished are those that follow the complete protocol, that is why the question "Is a mode-switch in execution?" is only asked for that protocol. In the other protocols a task could never be executed while a mode-switch is in a transition state, so there is no need to ask for other abort or suspend-resume protocols.

- `void vTaskChangeProtocolSwitchMode(short sNewProtocol, short sNewMode)`

This function is an easy way to change the protocol, and also to switch the mode. It is a combination of two functions: it calls the *vTaskChangeProtocol* function, passing argument *sNewProtocol* as a parameter, and then it calls *vTaskSwitchMode* to make a mode-switch to *sNewMode*.

- `void prvMoveTasksToNewMode(short sNewMode,subSCB *pxTempServer)`

This is an auxiliary function used from the *vTaskSwithMode* and *prvMoveCurrentServerCompleteProtocol* functions. Its goal is to move the server's task from the current mode to *sNewMode*. The procedure is as follows: it obtains the servers index using the *prsReturnAllArrayServersIndex*, then it goes through all the tasks in that server. If the task is the idle task then it removes its TCB from the ready list and it adds the task to the ready queue of the new mode. If the task is *not* the idle one, then it checks its behavior using the *xBehaviorMatrix* structure. If the task is inactive in the new mode, then the flag *uxIsSuspendedFlag* is turned to 'true' and passes to a new task. If the task is active in the new mode, then it saves the current location of the task (ready or release queue) and removes it from it. If the task was inactive in the old mode, then it updates the *xReadyTime* field and the value of *xGenericListItem*. The update is computed as follows:

```
difference = xTickCount - xModeTickCount[ auxTSK->sLastActiveMode ];
```

Where *xTickCount* contains the current time, and *xModeTickCount[auxTSK->sLastActiveMode]* gives the last time the task was active. Now *difference* is added to the old value of *xReadyTime* and *xGenericListItem*. If the task was active in the last mode, there is no need to update the values. Once the times are updated (or not) the system determines where the task must go. If the task was in the ready queue, then it must now go to the ready queue (updating also the *xReadyTime* to *xTickCount*). If the task was not in the ready queue, then another estimation is necessary to determine if the task must go to the delayed queue or to the

overflow queue. Consequently, it is needed to compute a safety margin that would be two times the server period in the task's last active mode. This means that the task should be executed at least once in the server's last two periods:

```
savePad = pxTempServer->xServerParameterList[auxTSK->sLastActiveMode].xPeriod*2;
```

The variable *savePad* stores this safety margin. Now, this margin is subtracted to *xTickCount* and the result is compared to the *xGenericListItem* value.

```
if(auxTSK->xGenericListItem.xItemValue > (xTickCount - savePad))
```

The explanation is as follows: the *xGenericListItem* value (hereinafter referred to as *wakeUpTime*) must know the next time that the task has to be “awake”, but perhaps the time selected coincides with a time when the server's budget is zero. This would cause the task to be "awoken" after its proper time. If this happens in a normal context, there is no problem because the system can wake up the task event if it is out of time. Now, an MCR is triggered (at time “t1”) before the server is executed and wakes the task up (“t1” it is bigger than the task *wakeUpTime*). Time after a new MCR is triggered (at time “t2”) to come back to the original mode; now the system is moving the task to the delayed or to the overflow queue. If this scenario (explained in FIGURE 10) occurs, then the updated value of *wakeUpTime* (*wakeUpTime'*) is smaller than the current time (even if a proper update has been made) but the task must go to the delayed queue. To avoid an improper allocation of the TCB it is necessary to compute a security margin. If the task is executed at least once in two periods of the server, the margin computed above will be enough to ensure that the task goes into the delayed queue (where it must go).

Briefly, this security margin ensures that the system keeps working properly even if a mode-switch occurs and a task is not executed for more time than its own period.

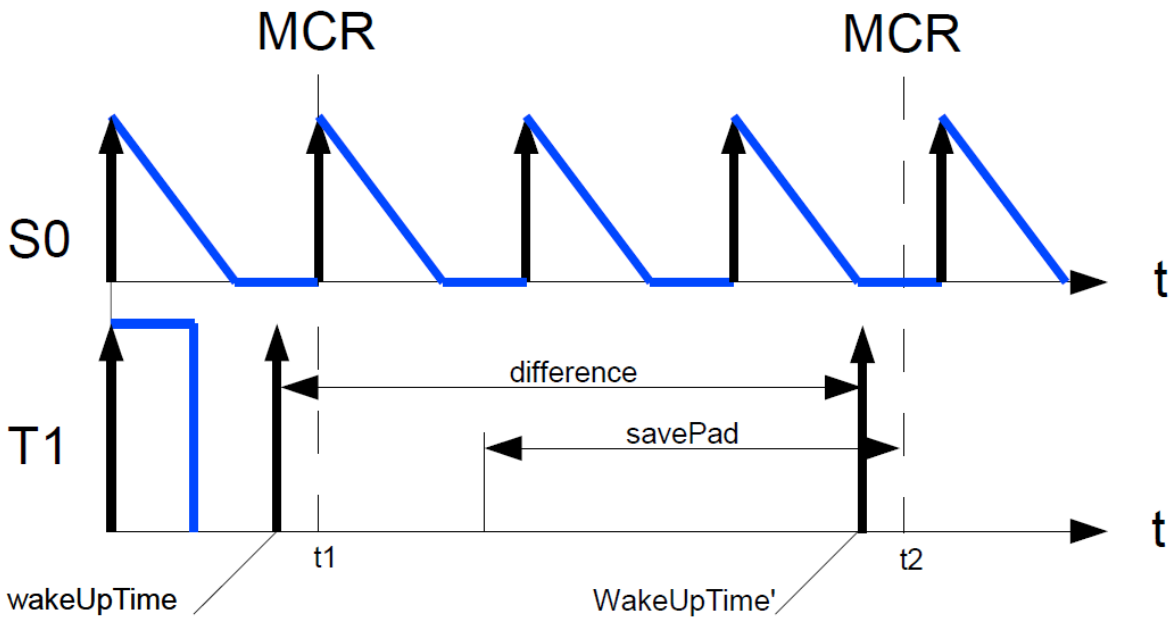

Figure 10: Usage of the savePad variable.

At this point the task is already located in the right place. It simply remains to update the *sLastActiveMode* field and to set *uxIsSuspendedFlag* to *false*.

- `void vTaskSwitchMode(short sNewMode)`

This function is responsible for the mode-switch from the current mode (also called old mode) to *sNewMode*. However, to perform a mode-switch, some conditions must be true:

- There should not be any other mode-switch in execution.
- A complete-protocol mode-switch is in progress. It means that a MCR has been made under the complete-protocol and there are other tasks in the current server that must be completed before the system finishes the mode-switch.
- *sNewMode* is smaller than *N_MODES* and it is not the current system mode.
- The MCR was not triggered during the *vTaskTickIncrement* or *vTaskSwitchContext* functions.

If any of these conditions are violated then the *vTaskSwitchMode* is finished without performing any change in the system. Mode-switch can start when all conditions are true. First of all, the interruptions are disabled so as not to disturb during switching. Then, the system starts to change the servers one by one from the old mode to *sNewMode*. To do this, the system goes through *pxAllServersArray* and performs the next procedure (except for the current server if *sSwitchModeProtocol* is set to *COMPLETE_PROTOCOL*):

(1) removing the server from any queue, changing the *sLocalCurrentMode* value to *sNewMode*,

(2) moving the server's tasks by calling the *prvMoveTaskToNewMode,* and

(3) finally, reallocating the server where it should be: if the remaining-budget is bigger than zero, then into the ready queue; if the remaining-budget is zero, then the system has to rely on the *uxInOverflowQueueFlag* to know where to put the server - into the delayed or into the overflow queue.

If the system is performing a mode-switch using the abort protocol then all tasks and all servers must go into the ready queue. This behavior is also followed in the *prvMoveTaskToNewMode* function.

Once the tasks are moved, then the system behaves in different ways depending on the protocol chosen:

- For the suspend-resume protocol, there is nothing more to do than simply restore the system to the proper task.
- For the abort protocol the only thing that remains is to set all servers' remaining-budget to its proper value (the server's budget value), by calling *the prvMoveCurrentServerAbortProtocol* function, and restoring the system.
- For the complete protocol, the procedure is more complex. At this point of the mode-switch procedure all servers were moved into the new mode queues, except the current server (*So*), the one whose task triggered the MCR. The system calls the *prvMoveCurrentServerCompleteProtocol* function, which turns on the *xCompleteFlag* flag and returns pdFALSE and sets the field *sIncompleteMode* to *sNewMode*. This makes the system restore the execution of the current task. When the current task reaches a “wait function” (e.g. *vTaskWaitForNextPeriod*) the system goes to the idle task of *So* and then executes the next task ready to run. When *So* has executed all its tasks, it comes back again to the idle task and now, the idle task calls the function *vTaskSwitchMode*, passing *sIncompleteMode* as the *sNewMode*. This makes the function go directly to the *prvMoveCurrentServerCompleteProtocol* function. This function moves *So* to *sNewMode* and returns *pdTRUE*. If an MCR is triggered from another task or from an interrupt subroutine during the mode-switch execution using complete-protocol, it is ignored until the mode-switch is completed.

At this point the three protocols reach the same point. First the system suspends the tasks by calling *vTaskSuspendAll*, then the *xModeTickCount* structure is updated properly and the field *sGlobalCurrentMode* is finally set to *sNewMode*. Now the interrupts are enabled, *xSwitchInCourseFlag* is set to *pdFALSE*, and the tasks are resumed. Then, if is necessary, the system is restored by calling the function *portYIELD_WITHIN_API*, which forces the system to select the proper *pxCurrentTCB* and execute it suddenly.

- `void prvMoveCurrentServerAbortProtocol(short sNewMode)`

The goal of this function is to set all servers' remaining-budgets to their proper values. Due to the abort protocol behavior, the proper value is the maximum they can reach in this mode (it means the server's budget), so a *for* statement is going through the *xAllServerArray* setting the remaining-budget to the server's budget according to *sNewMode*.

- `unsigned portBASE_TYPE prvMoveCurrentServerCompleteProtocol(short sNewMode)`

This function has two different behaviors depending on the value of *xCompleteFlag*. The first time this function is called, *xCompleteFlag* must be set to *pdFALSE*, then the system has to set the flag to *pdTRUE,* save the current time in *xCompleteDelayedTime*, and set the *sIncompleteMode* value to *sNewMode*.

The second time this function is called, then *xCompleteFlag* must be set to *pdTRUE*, which means that the complete-protocol mode-change is ready to finish. Then, the function moves the current server to the new mode as *vTaskSwitchMode* did with the other servers. At this point, a small trick must be performed. The system computes the time spent on completing the server execution and updates all tasks and servers different from the idle server and the current server. Finally, the function has to set *xCompleteDelayedTime* to zero, turn off the flag *xCompleteFlag* and return a *pdTRUE* value to let the *vTaskSwitchMode* finish the mode-switch execution.

It may occur that the MCR is triggered from an interrupt subroutine while the system is executing the idle task, which means that there are no other tasks to be executed. In this case *prvMoveCurrentServerCompleteProtocol* suddenly moves the current server to the new mode. And, since no task was executed during the complete-protocol mode-switch, there is no need to update the other servers.

5. Evaluation and results

This chapter explains the developing procedure. It explains the hardware platform, system testing and validations, and presents the behavior and performance results. In the end it presents the discussion on these results.

5.1 Work environment

Since our implementation is the extension of the HSF implementation, therefore, the same hardware and software platforms are used to develop this system as those used to develop HSF. The hardware used is a 32-bit board EVK1100 and the Dragon board as a debugger. Both are shown in Figure 11. The software employed was the integrated development environment (IDE) AVR32 STUDIO. And the operating system used is FreeRTOS.

- The EVK1100 board [9] is an evaluation and development kit from ATMEL. It is equipped with the 32UC3A0512 microcontroller and a wide set of peripherals such as parallel ports, led, buttons, Ethernet port, and an LCD display. The inside microcontroller is a low-power 32 bit with two memories of 512KB (flash) and 64KB (SRAM). The chips are programmed through the JTAG connector placed on the board.

§§

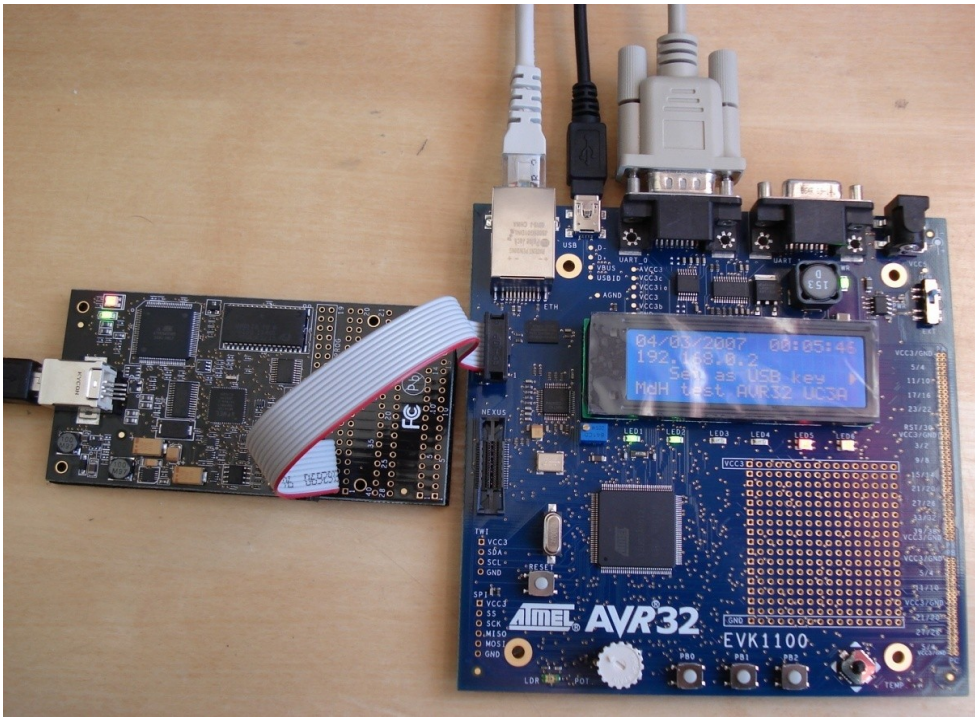

Figure 11: Dragon debugger and EVK1100 board.

- To download the code to the EVK1100 board, an AVR Dragon board from Atmel is used. This board is capable of not only downloading the code to the microcontroller, but also debugging the chip. It allows up to 32 software breakpoints and is able to read and write on the chip memory. This board is connected to the EVK1100 board by the JTAG wire, and connected to the workstation through a USB cable.

– The software employed to code, compile, program and debug the system was the AVR32 STUDIO, also from Atmel. This free software is an Eclipse based IDE designed to develop applications over Atmel devices, supporting a number of microcontrollers, boards and debuggers. The graphical user interface GUI is very friendly and the controls are very intuitive. The debug procedure is based on the *gdb* (GNU Debugger) and has the classic features: resume, suspend, terminate, step into, step over, step return, etc. Given the features of the board and the debugger there are two ways to see the variables' values: one is debugging the application and suspending it (with a breakpoint or directly with the "suspend" button), and then looking for the variable and its value. The other way is by using the USART through the serial port, however, in the machine where the work has been carried out, there is no serial port, so the only way to debug the system was to use the former.

5.2 Behavior evaluation

To validate the system it was necessary to prove (1) the correct behavior of the system during different modes and (2) to check if the different protocols are followed according to their desired behaviors. For this purpose two kinds of tests are done. One is made to pay attention to the tasks' behavior and check how the behavior changes among different protocols. The other one is made to prove the servers' behavior among the modes and with different mode change protocols.

The test was performed using a special function that is called at every system tick. This function stores the information about the current task and the current server that are being executed in a buffer. At the end of the execution (the execution was stopped when the tick count is about 200) the buffer is copied and presented in an excel document to generate the graphics.

The first test was performed with one server that executes two tasks in it. The server parameters are shown in Table 1. All values are constant in different modes. Since it is the only server in the system, the priority value is pointless.

Priority	1
Period	30
Budget	15

Table 1: Server parameters for the task behavior test.

This server contains two tasks. The first task, `Task 1`, executes an empty loop to consume CPU time and preempt itself, the second task, `Task 2`, executes an empty loop again to consume CPU time, preempt itself during its period and trigger an MCR. The tasks parameters are shown in Table 2. In this test, four modes are declared, and both tasks are active in all modes, but some parameters vary from one mode to another for `Task 1`.

	Task 1 in M0/M1/M2/M3	Task 2
Priority	3/3/3/3	4
Period	30/20/35/40	40
CPU time(in system Ticks)	9/7/6/1	4

Table 2: Task parameters for the tasks behavior test.

This test was performed for all protocols but the remarkable results are obtained in the abort protocol test and in the complete protocol test. Figure 12 shows the results for the abort protocol task behavior test.

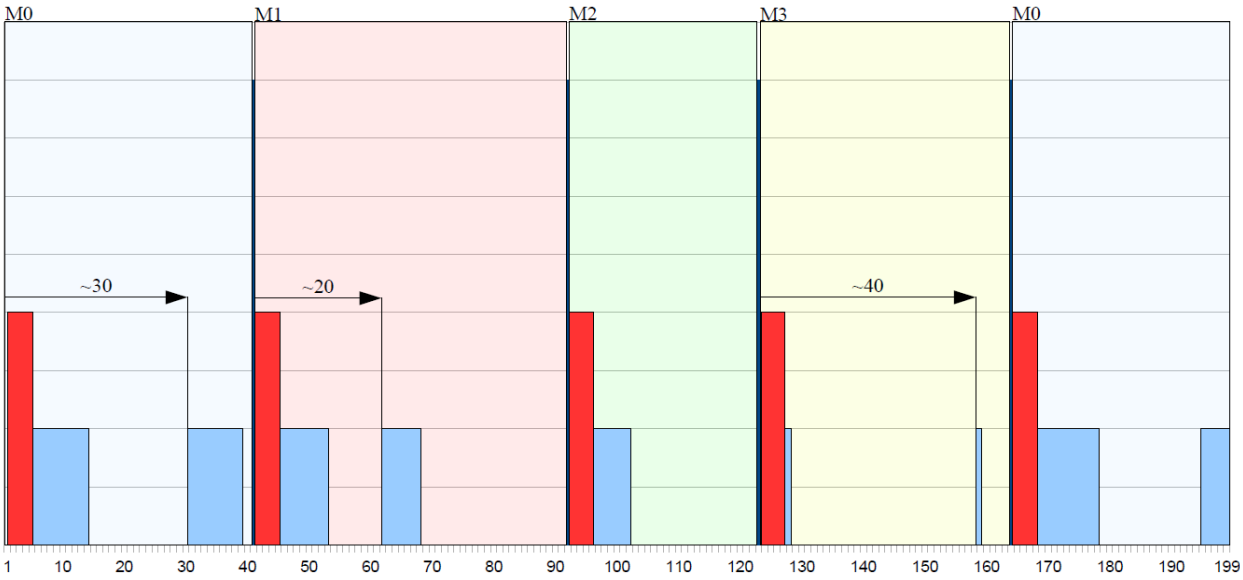

Figure 12: Abort protocol task behavior test results.

The red rectangles represent the execution of `Task 2`, the blue rectangles represent the execution of `Task 1`. The vertical lines through the graphic represent different mode change requests (MCR). The background color of the graphic vary to indicate the mode in which the system is: blue for *M0*, red for *M1*, green for *M2* and yellow for *M3*. The *X*-axis represents the system tick count. All these indications are valid for all the graphics in this section.

The way *task2* works is also the same for all the behavior tests: first the task consumes CPU time, then it calls a *wait* statement to "sleep" for "40" system ticks and, finally, when it is "awoken" it triggers an MCR. This is why all MCR seems to be made before *task2* execution, but they are the first thing that *task2* always performs.

In Figure 12, the behavior of the system is shown through different modes, making the mode-switch under the **abort protocol**. As can be seen, after every mode-switch both tasks (`Task 1` and `Task 2`) are executed. `Task 2` is responsible for generating the MCR and is executed every 40 system tics. However, for example, the third execution of `Task 2` is not produced at "80" (when it should). This is because of the server's budget and the task's period, i.e.; maybe the task is in the ready-queue but the server is still waiting for its ready time to come, so it is in the server's release-queue until "90", when the system activates the server and `Task 2` can then be executed. The *jitter* seen in Figure 12 is always caused by this asynchrony between the server and task periods.

The other interesting graphic is the result obtained from the **complete protocol**, as shown in Figure 13. Now, the vertical lines represent the MCRs, grouped into pairs. The first vertical line in the pair represents when the MCR is triggered, the second when it is finished. The narrow space between both is a transition state, where all the servers are in the new mode, "Mx", except the current server which remains in the old mode, "Mx-1", until all its tasks are completed. Often the server only has one task to be completed, for example, in the mode-switch from "M0" to "M1", where the system just has to complete the task that triggered the MCR. But sometimes the server has more tasks to be completed, for example, in the transition from "M1" to "M2". Here, *task2* triggers the MCR at the beginning of its execution, and once the task is completed the system switches to *task1* without switching the mode. Furthermore, when *task1* is completed and there are no other ready tasks in the server, then the system can finally switch completely to "M2".

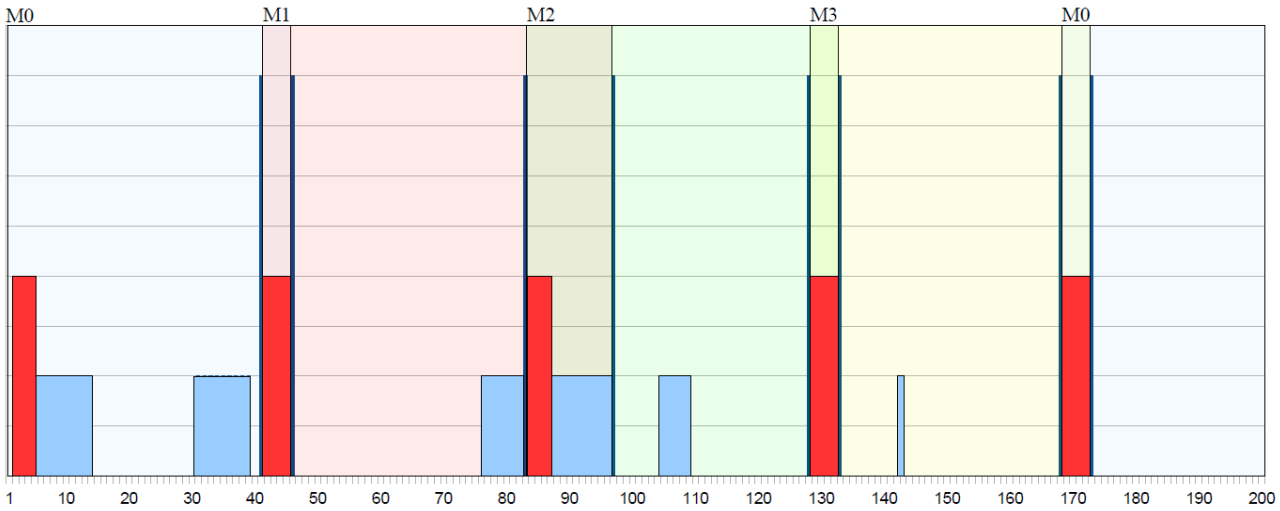

Figure 13: Results from the complete protocol task behavior test.

For the rest of the servers and their tasks, the time spent in the execution of the mode-switch is skipped, i.e.; all the times are updated and it is as if there were no transition states. If the current server's remaining-budget is expired during these transition states the system makes an exception and lets the server finish its execution properly.

The second test was oriented to observe the servers' behavior among different modes and with the different protocols. This test was performed with two servers with one task each. The servers' parameters are presented in Table 3 and the tasks' parameters in Table 4. For this test only two modes were employed to ease the understanding of the behavior. Unlike the first test, in this one the tasks have different behavior: *task1* is active for *M0* but inactive for *M1*; *task2* is active in both modes.

	Server1 in M0/M1	Server2 in M0/M1
Priority	2/2	1/1
Period	30/30	34/34
Budget	8/9	15/14

Table 3: Server parameters for the server behavior test.

The values of *task1* for the mode *M1* are set to 0, which means that *task1* is inactive for this mode.

	Task1 in M0/M1	Task2 in M0/M1
Priority	1/0	4/4
Period	30/0	40/40
CPU time(in system Ticks)	9/0	2/2

Table 4: Task parameters for the server behavior test.

The results of the test are very interesting for the three mode change protocols. The explanation order will be first the abort protocol, then the suspend/resume protocol, and finally the complete protocol. The graphs are formed by two color lines, yellow and orange. The first represents *server1*'s execution, and the second *server2*'s execution. Also, at the bottom of the graph there are the same blue and red rectangles as before, representing *task2* and *task1,* respectively. The *Y* axis represents the value of the remaining-budget for both servers: from 0 to

10 for *server1* and from 0 to 15 for *server2*. *Task1* belongs to *server1* and *task2* belongs to *server2*.

The first graph shown in Figure 14 is from the **abort protocol** behavior test. Because *server2* has higher priority than *server1, it* is executed first until its remaining-budget becomes 0. It is very clear how after the execution of the MCR all remaining-budgets raise again as the protocol ordains. Also, it is clear how *task1* (red rectangles) is executed only when the system is in "M0", being inactive during "M1". Please note that each server always contains an idle task also, which executes when there is no other higher priority task active in the server. Hence, in mode M1 the idle task of server 1 will only execute.

Finally, it is worth paying attention how, in this case, the period of *task2* is respected, executing the MCR every "40" ticks.

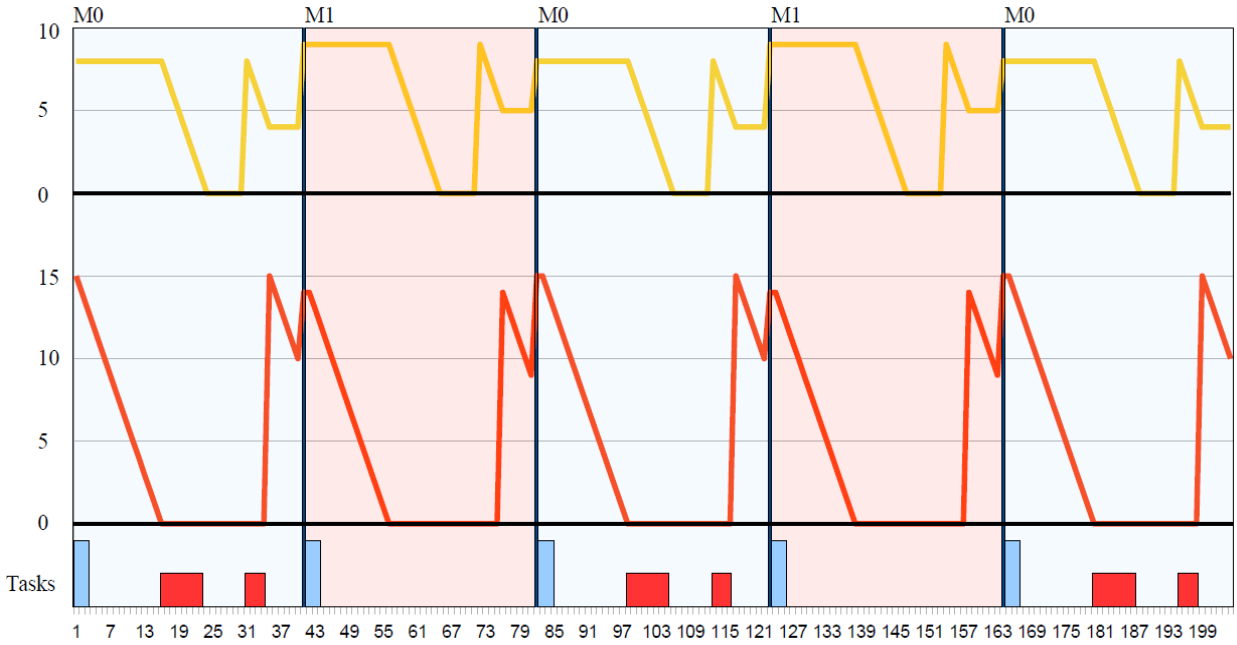

Figure 14: Results from the abort protocol server behavior test.

The second graph shown is from the **suspend/resume protocol** test (Figure 15). In this graph it is obvious that the remaining budget is restored from the value it encountered at the MCR. When the mode is changed from M0 to M1 for the first time, *server1* had a remaining budget of 4, and *server2* had a remaining budget of 10. When the system changes to the M0 at tick 80, *server1* and *server2* start their executions with the remaining budgets 4 and 10, respectively. According to the method used to measure the server's remaining-budgets, the graphic in Figure 15 shows the remaining-budget is 9. This is because the measurements are taken at every system tick, before the system decreases the remaining-budget. So, in proper terms, when the MCR is triggered at 80, the remaining-budget of *server2* is 9, and later in the next MCR the system restores the remaining-budget to 9.

This behavior is easy to understand by observing the *server2* execution in the second mode-switch to "M1". There is also a unique behavior in the last mode-switch. In all the figures, *task2* is executed after the mode-switch, but here it is done before and after the MCR. This is because *task2* does not complete the task in the previous execution; instead, it has to be preempted because the server's remaining-budget expires. Then, when the server is ready again, the task recovers its last state, completes the last execution and then goes to sleep until the next period is reached. However, "the next period" had already come while the server was in the release-queue, so the system executes *task2* again: first the MCR and then the CPU consumption time,

as it has always done. Also, it is worth highlighting how *server1* restores its remaining-budget after every MCR.

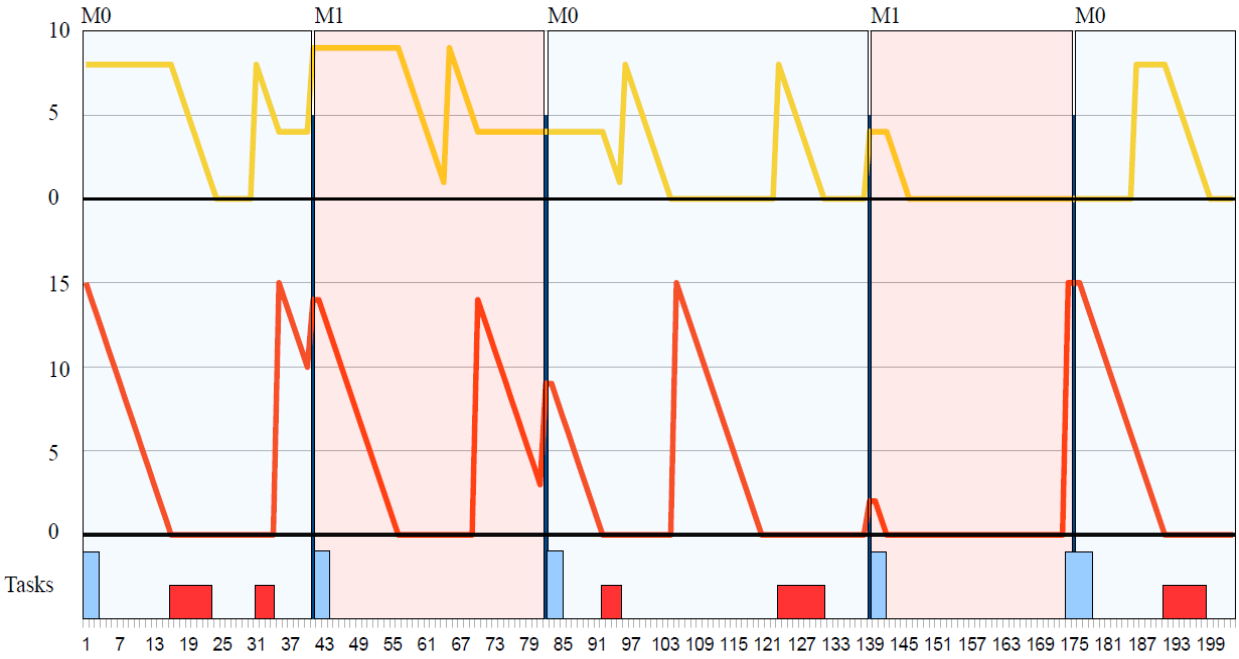

Figure 15: Results from the suspend resume protocol server behavior test.

The last server behavior test performed is for the **complete protocol**, shown in Figure 16. Due to the method used to obtain and represent the behavior of the system, it may seem to be a step in the scope of *server2*'s remaining-budget. Theoretically, this scope should not have these horizontals steps. As can be clearly seen in Figure 13, the MCRs are represented by a pair of verticals lines, the first is when the MCR is triggered and the second is when it is completed. During the transition state the current server (*server2* in the figure) consumes its remaining-budget until the tasks are completed (in the figure it is *task2*). Once the MCR is completed and all servers are in "Mx", the system restores the remaining-budget that the servers had the last time they was in "Mx". This behavior is clear in the first mode-switch from "M1" to "M0" in *server1*, restoring its remaining-budget from "4".

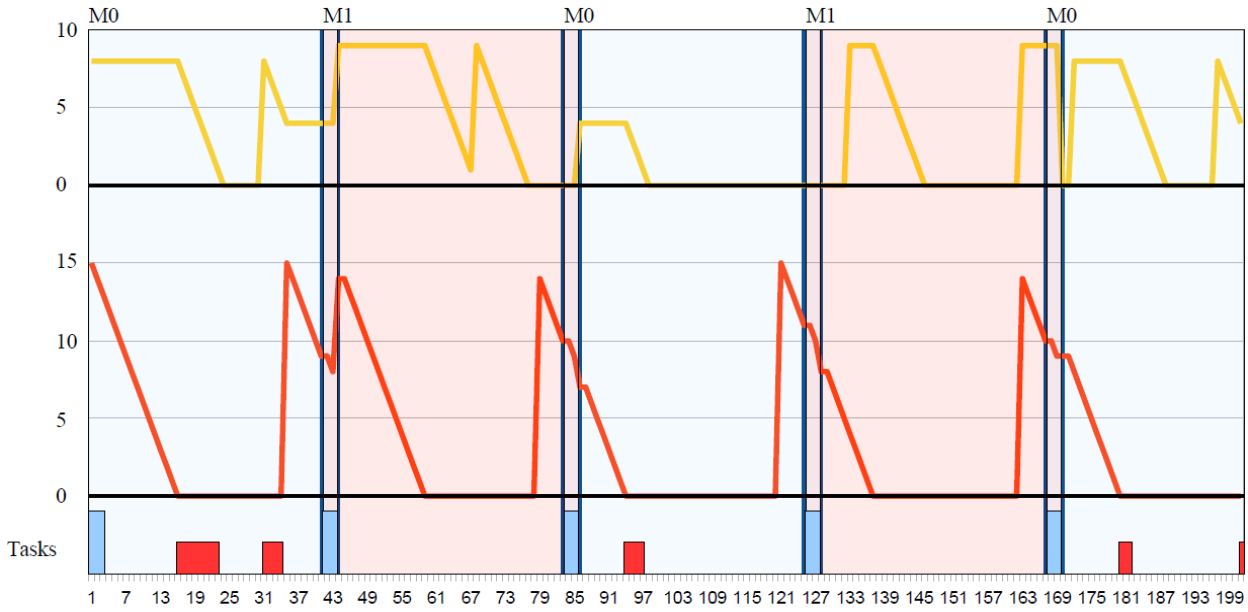

Figure 16: Results from the complete protocol server behavior test

5.3 Performance measurements

The objective of the performance test is to measure the time the system spends in the mode-switching procedure. This test has been done in several scenarios, varying the number of tasks and servers. Below, there will be an introductory explanation for each test that sets out the main features of each one so they are all understood properly, and the results are discussed in next section 5.4 - Discussion. All measurements shown are in microseconds (us), and the resolution of the system is 10 us. During the entire test the system goes through four modes, following the progression: *m0-m1-m2-m3-m0-...* The server parameters are not significant for this test because they do not affect the mode-switch behavior, only the complete protocol. However, the time values depend on what the CPU load of the task executed is within the transition. With respect to the complete protocol time values, there are two columns, the first shows the time spent in the second call of the *vTaskSwitchMode* function, i.e.; the time spent in changing the current server mode and to updating the others servers' and tasks' time values. The second column represents the whole time spent in the transition, since the MCR is triggered until the end of the transition. All tasks are active in all modes except for the final test.

I) First, results shown in Table 5 are from the "base test", which consists of 1 server with 1 task. This test shows the minimal values the system spends during the mode-switch.

	Abort	Suspend/Resume	Complete	Complete
Time Average	181,990	173,630	132,400	314,060
St. Deviation	3,164	5,198	4,989	5,510
Max Time	192	181	138	320
Min Time	181	170	128	309

Table 5: Time values for the first time test.

II) The next test consists of one server with two tasks. The transition time is increased by increasing the number of tasks in a server.

	Abort	Suspend/Resume	Complete	Complete
Time Average	208,490	197,800	159,450	2535,470
St. Deviation	5,437	4,960	2,409	3923,291
Max Time	213	202	160	9482
Min Time	202	192	149	330

Table 6: Time values for the second time test.

III) The third test consists of two servers with one task each.

	Abort	Suspend/Resume	Complete	Complete
Time Average	271,500	246,870	161,300	426,860
St. deviation	5,528	4,153	3,380	8,600
Max Time	277	256	170	512
Min Time	266	245	160	426

Table 7: Time values for the third time test.

IV) The fourth test consists of one server with four tasks.

	Abort	Suspend/Resume	Complete	Complete
Time Average	263,600	247,750	229,210	13634,620
St. deviation	4,292	4,787	20,634	1497,679
Max Time	266	256	256	15882
Min Time	256	245	202	11285

Table 8: Time values for the fourth time test.

V) The fifth test consists of four servers with one task each.

	Abort	Suspend/Resume	Complete	Complete
Time Average	455,400	420,200	219,360	646,750
St. deviation	4,408	4,960	8,984	8,972
Max Time	458	426	234	661
Min Time	448	416	213	640

Table 9: Time values for the fifth time test.

VI) The sixth test consists of three servers with three tasks each.

	Abort	Suspend/Resume	Complete	Complete
Time Average	508,750	478,990	266,180	3706,490
St. deviation	19,246	18,481	20,087	1427,190
Max Time	533	512	288	5856
Min Time	490	458	234	661

Table 10: Time values for the sixth time test.

VII) The last test performed tries to imitate a real scenario, where the task has different behaviors and different execution times in each mode. The test consists of two servers with 2 tasks each (from *task1* to *task4*). The servers' parameters are shown in Table 11 and the task behaviors in Table 12. The task periods and CPU times are of no interest as they only allow us to know that in *mo* the task that triggers the MCR spends a lot of CPU time after the mode-switch request. This CPU consumption is made to simulate a real scenario where the task responsible for triggering the MCR has also some other things to do.

	Server1	Server2
Priority	2	1
Period	20	40
Budget	10	15

Table 11: Server parameters for the real scenario test.

	Task1	Task2	Task3	Task4
M0	Inactive	Active	Active	Active
M1	Active	Inactive	Active	Active
M2	Active	Inactive	Active	Active
M3	Active	Active	Inactive	Active

Table 12: Task behavior matrix for the real scenario test.

The performance of the system for this scenario is shown in Table 13. At the bottom of the table are the values obtained in the first four mode-switches.

	Abort	Suspend/Resume	Complete	Complete	
Time Average	311,390	285,250	192,550	2036,180	
St. deviation	11,721	19,034	10,436	3004,754	
Max Time	330	309	202	9610	
Min Time	298	256	170	437	
Values (us):	298	256	181	9578	from M0 to M1
	298	309	170	437	from M1 to M2
	330	288	192	480	from M2 to M3
	309	288	192	458	from M3 to M0
	309	256	192	5642	from M0 to M1

Table 13: Time values for the last time test.

It can be seen how much time the system needs to spend in the transitions from *M0* to *M1* due to the large CPU load of the current server at the MCR moment.

5.4 Discussion

As for the behavior test, there is not a great deal to be reported. The graphics reflect that the system behaves as expected. Perhaps it may appear that there is a point where the system does not seem to behave perfectly. By observing the servers' behavior test, specifically from the abort protocol results, it can be seen how after the MCR, the remaining budget of *server2* decreases slightly. Ideally, after the transition under the abort protocol, the highest priority ready server must be executed, which would mean *server2,* but instead *server1* is executed. This instance explains why there are two different kinds of mode-switch behaviors presented for the abort protocol. The first has been used to carry out the behavior test. Once the system has completed the MCR and if a task is still active, it continues executing the server until the next system tick. That procedure is called the *soft ending*. The second is known as *hard ending,* and it forces a reschedule at the end of the *vTaskSwitchMode* function. That reschedule is done by calling the *portYIELD_WITHIN_API()* and will find the highest priority ready task in the highest priority ready server and set it as the current task. Which procedure to use can be configured in the *FreeRTOSConfig.h* file by giving the value "0" for the soft ending or "1" for the hard ending to the variable *HARD_ENDING*.

The *hard ending* procedure is also used when the task that triggered the MCR becomes inactive in the new mode. Both configurations work properly but have some minimal differences in performance; the *hard ending* procedure makes a reschedule which is expensive in terms of time. So this leads to a dilemma: the *hard ending* procedure is expensive in terms of time but follows the ideal behavior, while the *soft ending* procedure is better in performance but does not follow the right behavior.

So, **what procedure should be chosen?**

There is no correct answer to this question; the choice belongs to the users. Nevertheless, there are some factors that point to the *soft ending* as the better choice. To prove this, attention must be focused on the performance results. The first test simply measures the performance of an empty system and the aim of this measurement is to use it as a reference to know how performance varies for servers or tasks. The most interesting results are found from the second test onwards, where the average times go from 200 to 500 us. The system tick occurs once every millisecond; this means that the mode switch spends half a tick. Also, if a tick is reached during the transition, the system will automatically force a reschedule from the function *xTaskResumeAll()*. If the system tick is not reached during the execution of the mode-switch (i.e.; no system tick has been missed) it is probably close to being reached. Then, all mode-switches will conclude in one of the next 4 cases, depending on if the system tick has come or not during the execution of the MCR, and if the task has the highest priority or not. If the *hard ending* is being used, the system will probably behave in one of these ways:

- If a tick was missed during the mode-switch and the task has the highest priority, then the system will perform two reschedules consecutively and will not change the current task.
- If a tick was missed during the mode-switch and the task does not have the highest priority, then the system will switch to another task, and when the system comes back to the first task it will execute a schedule function again.
- If no tick was missed and the task does not have the highest priority, the system will schedule to another task, but not for an entire tick.
- If no tick was missed and the task is the highest priority, then the system will keep executing it.

Therefore, the only case in which a *hard ending* is useful is the third. Moreover, even if the

reschedule is justified as in the third case, the executing time until the next tick could be very short, depending on the amount of tasks and servers.

The discussion will now focus on the performance results. As expected, the mode switches following the complete protocol are the longest ones. Moreover, the abort protocol spent more time than the suspend resume protocol. That is due to the fact that in the abort protocol the remaining budget is restored for every server. The reasons are clear: looking at the base test, the difference in the times to change the protocols is due to the difference in the code used for the restoration of the remaining budget. In the suspend resume protocol some tasks go to the ready queue and others to the release queue. In the abort protocol all tasks go to the ready queue. This means that the macro *prvAddTaskToReadyQueue* is called more times. This macro spends more time to calculate the *savePad* variable and insert the TCB into the release queue. It is worth noting how the time difference between the protocols grows as more servers and tasks are involved. In fact this difference grows faster by increasing the number of servers than the number of tasks.

As regards the complete protocol times, there is nothing of particular relevance worth mentioning. By observing the second column (the one that represents the time spent in the whole transition), it can be seen how this time increases proportionally with the number of tasks in the server.

6. Conclusions and future work

6.1 Conclusions

The main goal of the project has been fulfilled: to develop a multi-mode hierarchical system and the different mode change protocols to switch from one mode to another.

The new system has been tested, obtaining results relative to its behavior, similar to the ideal described in chapter 3, and relative to its performance, obtaining acceptable values.

Also, the code generated has respected the preceding code, trying to modify it as little as possible. It is easy to configure and discard if necessary and has been fully commented to help future users and developers.

Finally, some additions to the explained code have been made to help future developers to expand the system.

6.2 Future work

The proposed future work is focused on the expansion of the assumptions, making the system more flexible and dynamic:

- To provide the servers the possibility of being inactive in some modes.
- To make it possible to share resources between modes.
- The capability of declaring new modes during runtime.

It is proposed to improve the current system for better performance:

- To provide the system with the capability to perform mode-switches during the increment-tick functions.
- To make it possible for an MCR to wait if another request is being executed.
- To solve the problem found with the long consecutive mode-switch requests.

Finally, it is proposed that a complete schedulability analysis be carried out.

7. References

[1] Yin Han, Hans Hansson and Etienne Borde. Composable mode switch for component-based systems. In the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011). 2011.

[2] Linh T. X. Phan, Insup Lee and Oleg Sokolsky. Compositional Analysis of Multi-Mode Systems. In the 22nd Euromicro Conference on Real-Time Systems (ECRTS10). July 2010.

[3] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Seyed Mohammad Hossein Ashjaei and Sara Afshar. Support for Hierarchical Scheduling in FreeRTOS. In Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 11), pages 1-10. September 2011.

[4] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin and Moris Behman. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In Proceedings of the 7th International Workhop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11), pages 51-60. July 2011.

[5] Rafia Inam, Mikael Sjödin and Reinder J. Bril. Implementing Hierarchical Scheduling to Support Multi-Mode System. In the 7th IEEE International Symposium on Industrial Embedded System (SIES' 12), WiP. June 2012.

[6] Nathan Fisher and Mased Ahmed. Tractable Real-Time Schedulability Analysis for Mode Changed under Temporal Isolation. In the 9th IEEE Symposium on Embedded System for Real-Time Multimedia (ESTIMedia 11). October 2011.

[7] Ying Han and Hans Hansson. A mode mapping mechanism for component-based multi-mode system. In the 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2011), pages 38-45. November 2011.

[8] FreeRTOS web-site: www.freertos.org .

[9] ATMEL EVK1100 and AVR32 Studio software documentation: www.atmel.com/tools/EVK1100.aspx?tab=documents .

[10] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In IEEE Real-Time Systems Symposium (RTSS'97), 1997.

[11] K. Tindell, A. Burns and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In Real Time Systems Symposium (RTSS), 1992.

[12] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In 10th Euromicro Conference on Real-Time Systems. 1998.

[13] V. Neils, B. Andersson, J. Marinho and S. M. Peters. Global EDF scheduling of multimode real-time systems considering mode independent tasks. In 23rd Euromicro Conference on Real-Time Systems. 2011.

[14] X. Ke, K. Sierszecji and C. Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA-07). 2007.

[15] E. Borde, G. Haik and L Pautet. Mode-based reconfiguration of critical software component architectures. In Conference on Design, Automation and Test in Europe (DATE 09), pages 1160-1165. 2009.

[16] P.H. Feiler, D. P. Gluch, and J.J Hudak. The architecture analysis and design language (AADL): And introduction. Technical Report, CMU/SEI-2006-TN-001. 2006.

[17] T. A. Henzinger, B. Horowitz and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In PROCEEDINGS OF THE IEEE, pages 166-184. 2001.

[18] J. Templ. TDL specification and report. Technical Report, Univ. of Salzburg. 2003.

[19] Ben-Ari, M., "Principles of Concurrent and Distributed Programming".ISBN 0-13-711821-X. Ch16, Page 164. 1990.

[20]http://www.thefreedictionary.com/unimodal

[21] Inc. McGraw-Hill Dictionary of Scientific & Technical Terms, 6th edition, 2003. By The McGraw-Hill Inc.

Appendix A: API

The new private functions are:

- `short prsReturnAllServersArrayIndex(subSCB *pxServer);`
- `short prsReturnTaskArrayIndex(tskTCB *pxTCB);`
- `void prvMoveTasksToNewMode(short sNewMode,subSCB *pxTempServer);`
- `void prvMoveCurrentServerAbortProtocol(short sNewMode);`
- `unsigned portBASE_TYPE prvMoveCurrentServerCompleteProtocol(short sNewMode);`

The new public functions are:

- `unsigned portBASE_TYPE xTaskChangeServerModeBehavior(short mode, unsigned portBASE_TYPE xBehavior);`
- `unsigned portBASE_TYPE xTaskChangeTaskModeBehavior(short mode,unsigned portBASE_TYPE xBehavior);`
- `void vTaskStartModeScheduler(short defaultMode);`
- `void vTaskChangeProtocol(short sNewProtocol);`
- `short sTaskGetCurrentSystemMode(void);`
- `portBASE_TYPE xTaskIsCompleteInCourse(void);`
- `void vTaskChangeProtocolSwitchMode(short sNewProtocol, short sNewMode);`
- `void vTaskSwitchMode(short sNewMode);`

A detailed explanation for all these functions can be found in section 4.3.