ADAPTING MODE SWITCHES INTO THE HIERARCHICAL SCHEDULING

DANIEL SANCHEZ VILLALBA
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 is said to operate in multiple modes where each mode corresponds to a specific application scenario, are called Multi-Mode Systems (MMS). Normally, a different piece of software is executed for each mode. At a specific time, the system can be in one of the predefined modes and is switched from one mode to another upon some condition. A mode switch mechanism (or mode change protocol) is used to transform 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 in this thesis is the extension of the HSF with the multi-mode feature with the emphasis of doing 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. The implementation now supports different modes of the system which can be switched at run-time. Each subsystem and task exhibit different timing attributes for different modes, and upon a Mode Change Request (MCR) the task-set and timing interfaces of the whole system (including subsystems and tasks) are changed. A Mode Change Protocol specifies the way to change the system-mode. 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 is changed. While changing a mode from normal to emergency may require aborting all the 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, Abort protocol and Complete protocol. These protocols increase the flexibility of the system, letting the users to select the way they want to switch to the 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 mode of the system 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

0. Abstract .............................................................................................................2

1. Introduction .....................................................................................................4
   1.1 Real-Time System .......................................................................................4
   1.2 Multi-Mode System and Mode switches ......................................................5
   1.3 Related work ...............................................................................................6

2. Background ......................................................................................................7
   2.1 Real-Time System and Real-time Operating System ..................................7
   2.2 FreeRTOS ...................................................................................................8
   2.3 Hierarchical Scheduling Framework and its implementation on FreeRTOS 9

3. System Design ..............................................................................................11
   3.1 Assumptions ..............................................................................................11
   3.2 System model ............................................................................................12
   3.3 Mode change protocols .............................................................................15

4. Implementation ............................................................................................19
   4.1 Data structures ..........................................................................................19
   4.2 Modified API and Macros ...........................................................................23
       4.2.1 Modified Macros ................................................................................23
       4.2.2 Modified API ......................................................................................24
   4.3 New API .....................................................................................................28

5. Evaluation and results .................................................................................33
   5.1 Work environment ......................................................................................33
   5.2 Behavior evaluation ...................................................................................35
   5.3 Performance measurements .....................................................................41
   5.4 Discussion .................................................................................................44

6. Conclusions and future work .................................................................46
   6.1 Conclusions ...............................................................................................46
   6.2 Future work ...............................................................................................46

7. References .....................................................................................................47

Appendix A: API ............................................................................................49
1. Introduction

The complexity and size of the real-time embedded system software is increasing day-by-day. Usually they are required to provide a wide variety of application scenarios in the same system. The very different and changing application scenarios not only increase the overall complexity of the real-time embedded systems, but it also demands more coordination and management among different functions of the system. Moreover, a dynamic change in the application scenarios is also required that usually change the behavior and services demanded by the user at runtime. All these challenges require a methodology to handle the complexity of the system and to provide good results to the user, which is difficult to develop without spending long time and great resources.

One way to avoid that expensive development is simplifying the system: not by restricting the services, but by dividing it into different parts so that the development and maintenance of those parts becomes easy, and later those parts are combined together to form a 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 development of a Multi-Mode Hierarchical Scheduling Framework (MMHSF).

1.1 Real-Time System

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

Example: An Airbag system:

An Airbag system in the car is a classical example of the real-time system where the timing constraints play an important role. If a car has an accident, the airbag system must ensure the occupant’s safety. Because of this, the airbags must be inflated quickly; otherwise a life could be lost. There is a specific response time for an airbag system fixed in 1 ms (millisecond). So the embedded system responsible for the airbag deployment must spend less than 1 ms in the response. This is a hard real-time-constraint, if it is not achieved, a life would be lost. For this purpose, a real-time system is used, because it is going to guarantee that the response time (the time since the car realizes it has had an accident, until the airbag starts deploying) is less than 1 ms. This time limitation is called “time constraint” and it has to be declared for every task in a real time device.

In summary, Real-Time systems are those systems that guarantee the performance of the tasks within the specified time. This feature makes the real-time systems very accurate time devices, often used to accomplish critical tasks that should not exceed a certain deadline. This means that a delay in the task's execution could cause a severe damage or failure (e.g. airbag system or the car's ABS). Real-Time systems are also used in high performance applications, where the quality of services it 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

Typically systems are 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 manage its behavior according to different external or internal condition. For example, consider a device powered by a limited battery resource, charged under normal conditions and behaves as a uni-modal system. At some point when the battery gets down, the device should manage itself by reducing the power consumption; for instance by reducing the screen’s brightness or the processor load, etc. Each of these services needs to realize about the battery level and manage itself to modify its behavior accordingly. Moreover in the given example, there should be a battery module responsible of keeping track of the battery level, and other modules (like screen or processor management). Other modules should ask the battery module to know about the actual level of charge in the battery. The system could be in normal mode when the battery is full, and could be in the low battery mode otherwise. This example indicates the need to change the system’s mode dynamically depending on the battery situation.

A system that operates in different modes and each mode has a particular functionality and a different timing behavior is called a Multi-Mode System [6]. The system realizes about the conditions and switches from one mode to another mode at run time. Its tasks modify their functionality and timing behavior upon a mode change. A Mode Switch Mechanism (or Mode Change Protocol) is used to transform the system from one mode to another at runtime [6].

Getting back to the example explained above, the device will behave as a multi-modal system. When the battery drops below a certain threshold, the battery module will realize it, and notify it to the system. The screen will now notice that the battery is low, and provides a signal/message to the system. Then the system will switch the mode, for instance, from “normal mode” to “low battery mode”. This mode switch will make the services and modules to vary 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 a multi-mode one. But this process is not as simple as it seems. The above example is simplified to ease the understanding of the multi-mode systems, but there are many questions related to this: How quickly should the system switch to the new mode? How to notify the new mode to the tasks? And what would happen if some tasks have nothing to do in the new mode? All these questions have been investigated since long and multi-mode is nowadays a well known technique used in embedded systems. But those questions haven’t been investigated and applied to the implementation of simple hierarchical system (HSF).
1.3 Related work

There is no work done with respect to the implementation of a multi-mode hierarchical system. A multi-mode schedulability analysis is presented in [11][12] and [13], and another analysis about a compositional system in [2], this last paper presents a multi-mode model and some techniques for analysis systems that contains various applications. It has also presented 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 a lot of ideas to develop the MMHSF.

Some works about multi-mode frameworks are presented in [14] and [15], where methodologies focused on design reconfigurable, critical and complex embedded systems are presented. There are some others papers that talk about programming languages that support multi-mode, presented in [16][17][18].

A detailed Mode Switch Logic (SML) algorithm for multi-mode on component-based system is given in [7]. This SML implements coordination and synchronization of mode switch in a component-based systems. This logic is implemented under the assumption of that all the components support the same modes, but a way to overcome 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 developing the ideas from the SML presented in [7].

Finally a generic framework to implement a Multi-Mode Hierarchical system has been presented in [5]. It is based in a two-level HSF implementation on FreeRTOS and provides a framework to change the system from uni-modal to a multi-mode one. It proposes the initial design details for the MMHSF implementation with the aim of doing the minimum modifications 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. And then we present three different mode-switch protocols to change the system mode and their implementation details.
2. Background

This chapter explains the background technologies upon which our work is based on for better understanding of our work. The first section explains real-time systems and goes into the issues of a real-time operating system (RTOS). The second section gives a general vision about a concrete real-time operating system: FreeRTOS upon which this implementation is done. The chapter ends up with a brief explanation about the Hierarchical Scheduling Framework.

2.1 Real-Time System and Real-time Operating System

For those outside the electronics and computer science, 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 some purpose [21]. As it has been discussed, a real-time system is the one which ensures that its tasks will execute within its time constraints. This feature is controlled by the operating system that governs the framework, called real-time operating systems (RTOS). The RTOS is responsible of guaranteeing the execution of all tasks in timely manner. To accomplish this goal, there are some features that allow the RTOS to ensure the time-constraints:

- An RTOS must be completely aware of the time outside the system (means the “real time”). As it has been told 1 ms in the system must be a real 1 ms.
- It must be fast in switching from one task to another, spending as less time as possible in the task context-switch.
- The system must have some sort of interrupt subroutines, giving the control of the execution to the scheduler as soon as possible.

All 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 to be executed when and for how long. Generally, RTOS schedulers have two main policies:

- Preemptive Priority (also called 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 variables.
- 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 good algorithm to execute them. This should be a deterministic algorithm, means that for a given input it always behaves in the same way. The most deterministic the algorithm is, the most predictable the system is. But sometimes this is not enough due to the executed tasks which often are non-deterministic. This leads to one of the problems in RTOS, the jitter. Jitter can be explained as the deviation in the executing time spent from the ideal executing time. The jitter phenomenon is well known, and schedulers keep it in mind. But sometimes a task needs more time to be executed and it is not possible, so the task would not accomplish its deadline. That may cause different effects depending on the type of the 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 although it is not as good as the result computed within its deadline.

We are considering periodic execution of the 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 the responsible of 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 wait until the next period. This is done by the task by calling a \textit{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 record of the time to activate the task again and there is no need of additional structures to save the task status. In the second method the scheduler keeps track of nothing, instead it requires to save the state of the task in some place (usually status registers) to restore it when it is necessary. We are using the second approach in our work.

\section{2.2 FreeRTOS}

FreeRTOS is an open source real-time operating system \cite{8}. It is developed by “Real Time Engineers Ltd.” mainly in C language and supports 31 different hardware architectures. It’s very simple, easy to use and modify. Its scheduler runs at the rate of one tick per milli-second by default, but it can be changed to any other value easily by setting the value of \texttt{configTICK\_RATE\_HZ} in the FreeRTOSConfig.h file.

FreeRTOS’s 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 its execution. At this moment the system saves the current state of the task in a structure called task control block \texttt{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 the \textit{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 \textit{ready queue}, where the tasks are placed when they are ready for the executed. The ready queue is an array of \texttt{xList} elements that behave as an ordered queue, sorted according to the tasks priorities.

2. The second queue is a \textit{release queue} where the tasks go when they have been executed (when they are preempted). It consists of a \texttt{xList} elements that sorts the tasks by its 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 \texttt{idleTask}. This task is automatically generated by the operating system, cannot be modified by the user, has the lowest priority and never calls a \texttt{wait} function.

The system has a hardware timer continuously counting the time. Every millisecond (ms) the system tick increments its time, storing the current time in a field called \texttt{xTickCount}. At each
system tick, the scheduler checks the release queue and checks the first task, if its wait time has expired then move the task from the release to the ready queue and checks the second task; if not then continue its common 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 the field pxCurrentTCB). If the new task has higher priority than the currently executing task then a switch context is made (the current task stop its execution and save its current state into its tskTCB field, then pxCurrentTCB is pointed to the tskTCB field of the new task and the system restore the last state of the task stored in pxCurrentTCB).

The way to make run the FreeRTOS is to modify the main.c file, creating all the tasks, declaring these tasks into 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 upon 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 manages several tasks in it. These servers are scheduled by a global-level scheduler that governs the whole system. Each of the subsystem has its own ready and release-queue independent from others subsystems. These subsystems (servers) are like the applications on 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 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 its 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 to 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 idling periodic server type, whose execution process is explained below.

**Example: Idling Periodic server Execution**

Consider two servers, So and S1, as Figure 1 illustrates. The So has the higher priority than the S1, and both have different periods To and T1 respectively. The arrival time for both servers is represented by an up arrow. So has a smaller period than S1, and also a smaller budget (the budget is represented by the arrow’s height). As it can be seen from Figure 1, at the start both servers want to execute, they have the remaining budget more than zero. But since they cannot be executed at the same time, the highest priority server So is executed first. The blue line represents the server execution. As time passes, the remaining budget decrements and eventually reaches 0. At this point of time, all tasks in So are preempted and a context switch is made by the system, changing server from So to S1. Now S1 starts execution and its remaining budget starts to decrease. At time To the server So will be activated again, because it’s period has expired, returning the remaining budget value from 0 to the budget value. Since So has higher priority than S1, it causes another context switch, from S1 to So. It is worth noting that S1 has been interrupted in the middle of the execution, and its remaining budget is not 0. When So will expire its budget, it will be preempted and S1 will start its execution again from the exact moment where it was interrupted last time. S1 will finish its execution when its remaining budget expires.
There is a concrete moment in Figure 1, when both servers have their remaining budgets equal to 0. What is happening in the system? Neither S0 nor S1 are executing, then: What is the system executing? In this case, when all servers have their remaining-budget equal to 0 then the system will execute the idle server. The idle server is a special server, automatically generated by the system at the start of the execution, when the function vTaskStarScheduler() is called. This server has the lowest priority, i.e. 0, and infinite period and budget. So 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 the 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 others servers have). There is no way that a user can create a new task inside this server. Its function is to keep the system running when others servers have expired their remaining-budgets.
3. System Design

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

3.1 Assumptions

The assumptions are some barriers to limit the scale of the design, just to be concrete what it should perform. Later some of these assumptions could be relaxed or changed by other less restrictive ones to make the design grow.

The basic assumptions are:

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

II) No shared resources between subsystems and modes with the abort protocol. This assumptions will ease the 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 doesn't vary from one mode to another; always work in the same way.

IV) Same task behavior. The task behavior (functionality and timing properties) remain the same in all modes, just can select if 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 all 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’s 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.

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).

Further, the system has several modes that determine the subsystems’ and tasks’ behavior, being able to switch from one to another. Changes from one mode to another are managed by the Mode Change Request Controller (MCRC), which is responsible of capturing a request to change the mode (made by a task) and communicating it to other MCRC in the system. This mechanism is performed in an hierarchical manner, i.e. a global MCRC receives an Mode Change Request (MCR) from a task within the server. The local MCRC transmits this request to the global MCRC. Then, the global MCRC notifies the others local MCRCs to change the mode of
the servers. The mode indicates the current context of servers and tasks. As it has been seen in hierarchical scheduling framework section, each server has its associated timing parameters called timing interface (period, budget and priority). In the multi-mode system these timing interfaces are defined for each mode separately, being able to be different from one mode to another. The same happens to the tasks, they can have different timing properties in each mode.

This briefly explains the behavior of the system to change the mode: by switching the local modes of every server. Then it can be derived 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 one, except in the transition state, where they can differ.

To switch from one mode to another a task must trigger a Mode Change Request (MCR). MRC is the mechanism to change the system's mode. The MCR is a demand 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 to perform the mode change. The server that triggers the request must behave depending on the protocol specified by the trigger function. The transition state is the state of the system during which the system is changing its mode from old mode to the new mode, and the system schema during transition state can be shown as Figure 3.

![Figure 3: System schema during transition state](image)

The task $T_0$ of server $Server_0$ trigger the MCR. Instantaneously the local MCRC forwards the request to the global MCRC. The global MCRC then communicate this request to the local
MCRCs in other servers (Server1 in Figure 3) to automatically change its mode (to Mode2). At this point the whole system is in Mode2, except Server1 that is still in the previous mode Mode1.

At this point we could have two different scenarios (according to the 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 needs some external help to be solved. At this point the other parameters of the MCR come into play: the protocol and the deadline as explained in the next section.
3.3 Mode change protocols

An interesting question was made in the previous section: What happens with the task that triggered the MCR? A brief answer was made and detailed explanation is as follow:

Focusing on the first scenario described (the task is active in both modes) there can be two cases: task has the highest priority in the new-mode, then the system will continue executing the task; or the task does not have the highest priority, then the system will suspend the task (as if the task reaches a `wait` statement) and will add it to the ready-queue based on its priority.

Focusing on the second scenario (the task is active in the old-mode but inactive in the new-mode), what should happen 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 as explained below:

- **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 releases the possible shared resources it had locked. When the system comes back 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, switch to the new-mode and, when the system comes back to the old-mode again, it resumes those tasks from the point where they were previously suspended.

The two first protocols Complete- and Abort-protocol are mostly clear in its functionality: to let complete all tasks till the end or to stop the execution of all tasks at once respectively. But in case of Suspend/resume-protocol there are some questions that do not have a clear answer and it is worth being discussing them.

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

As explained in the hierarchical scheduling section 2.3, each server has a remaining budget which is equal to the servers's capacity/budget at the start of the server execution and decrements when the server executes. In the multi-mode context, each server has 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 of 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 the time zero, therefore 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 a first MCR from Mode0 with suspend/resume protocol specified, and a second MCR with abort protocol to Mode0, what should happen?
Using suspend/resume protocol, the system suspends all tasks and servers of old-mode in the system and then resumes all tasks and servers of new-mode. When the tasks and servers are suspended, their status for old-mode is stored and when this old-mode comes again, whatever is the second MCR protocol, it will resume the task and the servers move all to the ready list (ready task list for tasks, and ready server list for servers).

### III) What should happen to the task in release queue at an MRC request? When should the task be activated?

As it is known, the tasks are self-triggered, i.e. each task indicates to the scheduler when it wants to be “activated” again by a time-based wait statement. In the HSF implementation, this statement makes the system to move the task to the release queue (from the ready queue) and when the specified time has passed, then moves it back to the ready queue.

In the MMHSF system, it may happen that an MCR is triggered during a task is waiting in the release queue. Suppose that this task is inactive in the new mode. Then the system will keep track of the actual time the task was waiting until the MCR was made, computes the remaining time the task has to wait and saves 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 illustrate how this activation is made. In mode M0 when the first MRC is made, the task T0 should be activated after 2us. This time is stored in the system when the mode is changed to the M1. Later at the second MRC request when the system changes its mode back to M0, the task T0 is activated after 2us.

This method is called frozen-time and it works 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 into a structure that saves the time when an MCR is executed by the system (called xModeTickCount).
When a new MCR is triggered to restore the system to the old mode (in the diagram is the arrow “t2”), the system has to compute for how long the task must keep sleeping. 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:

\[ \text{diff} = t2 - t1 \]

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:

\[ t'' = t' + \text{diff} \]

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, then just remain to move the task into the pxDelayedTaskQueue.

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

Suppose the Figure 6 context, in which a server S1 (with the highest priority) has spent all its budget and server S2 (lowest priority) suffers an MCR during its execution. The remaining budget of S2 will be saved in the timing interface field commented above (the same is done for S1, but 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, 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. But in this scenario both servers are inactive in M1, so the keep waiting without being executed. After some time another MCR is made to change the system to mode M0. At this point the system will find with four different kinds 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. There is nothing to do with the first and second kind. The third kind was explained in the first part of this example. The interesting procedure is for the fourth kind. We can split these servers into two sub-kinds: 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 came, i.e. they were in the release queue. For the first type the procedure is simple: just to restore the old remaining-budget (which was stored previously, in the first MCR) and move them to the current ready-queue. The procedure for the second type is more complex: the system has to compute how long these need to keep waiting, in order to accomplish the period constraints, and saves this time.

Figure 5: Frozen-time procedure.
into the $x_{\text{ReadyTime}}$ field. Finally the system has to move the servers to the release-queue. But, since the VI assumption requires all the servers to be active in all modes, this procedure is not employed and implemented (but the code is already prepared to support this feature in future).

It is also **frozen-time** and it works as same as **frozen-time** that works for tasks. Since servers are always active (Assumption VI) there is no need to think about what would happen to a server.
4. Implementation

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

4.1 Data structures

To fulfill 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 of the 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 last field that gives the last mode in which the task was active(*sLastActiveMode*) as shown in Figure 7. Also, 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).

Some new variables and structures are required to make the system work properly and 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). This time, combined with the tskTCB field that informs about the last mode the task was active is very useful to compute how long the task has been inactive, as it was explained in the previous chapter, in the frozen-time explanation.
- 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 it can be seen that the changes performed since the HSF implementation and how the servers’ queues are now two-dimensional queues.

It is also necessary to declare a new structure that contains information about all servers and all task 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 the system spends 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 queue. This structure is modeled as an array, one element per server with the next fields and is shown in Figure 9.

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

Because 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 may have some procedure to erase a task from the tasks array. The way to make an array dynamic is to declare a pointer of the elements type. For this purpose two new types are declared. Firstly, the \texttt{taskArrayElement} type, that is a pointer of type \texttt{tskTCB}. With that type, a pointer to a \texttt{taskArrayElement} can be declared which means a double pointer to a \texttt{tskTCB} structure, that also means that a new array structure has been created, an array where the elements are \texttt{tskTCB} pointers. In this new type, all the tasks contained in a concrete server can be grouped together. So a second structure is required, one that contains the array commented above and the \texttt{subSCB} pointer of the server to which the tasks of the array belong to. That structure is called \texttt{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, \texttt{pxAllServersArray}: a pointer of type \texttt{serverArrayElement}. This pointer is a dynamic array that allows using the functions \texttt{pvPortMalloc} and \texttt{pvPortRealloc} to dynamically allocate and deallocate servers in the system. The same functions are used to manage the \texttt{taskArrayElement} pointer that contains all the tasks in a concrete server.

![pxAllServersArray structure](image)

\textit{Figure 9: pxAllServersArray structure.}

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

With all these variables and data structures now the system is capable of sustaining different modes within hierarchical scheduling, it just needs the procedures/function to manage them properly. 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 these modified or newly created routines are explained.

From this point, this chapter describes 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 done 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 to ease the mode-switch mechanism or the whole system performance 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 \texttt{configMULTI_MODE} value to zero in the \texttt{FreeRTOSConfig.h} file. We use compiler directives like \texttt{#if(configMULTI_MODE)}, \texttt{#else} and \texttt{#endif}. If \texttt{configMULTI_MODE} is set to 1 then the constant \texttt{N_MODES} must be set to higher than 1. \texttt{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.

Now 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 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 now can pay attention just to the server's remaining-budget to 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 the 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 the particular mode.

- **prvAddServerToReleaseQueue( pxSCB )**
  inserting the server into the release queue. Release queue is a priority array sorted according to the next activation time of the server (\( x_{ReadyTime} \) of the server).

- **prvAddServerToOverflowReleaseQueue( pxSCB )**
  inserting the server into the overflow release queue. Overflow release queue is a priority array sorted according to the next activation time of the server (\( x_{ReadyTime} \) of the server).

In all of these macros the value of the flags \( ux_{InReadyQueueFlag} \) and \( ux_{InReadyQueueFlag} \) is updated properly to the destination of the server.

- **prvAddTaskToReadyQueue( pxTCB )**
  In this macro the task is inserted in the server ready queue sorted by its priority. Also update the \( ux_{TopReadyPriority} \) if it is necessary. To make this function works properly it is necessary to correct the access to the new priority array and add the task to the right queue, sorted now according to both priority and mode.

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

- **prvCheckDelayedTasks(pxServer)**
  This macro looks into the release queue of the server and finds the task whose \( x_{ReadyTime} \) has expired, and adding it to the ready queue. Since the \( pxDelayedTaskList \) is an array this macro it has been changed to ensure the proper access to the queue.
4.2.2 Modified API

Functions are sorted by the order of appearance in the code. As it has been explained before most of the changes consists into 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 in the new code this structure is two-dimensional array it needs a nested for structures. Similarly others 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 compute, 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 concerned to the ready and release queue and to the xServerParameterList. Further, this function was responsible of setting the remaining-budget when the server goes to the ready queue. To change that it was required to reallocate a line of code. This 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. The functionality is to move pxServer to the proper list and update the next ready time of the server. Since the system is built to support shared resources this function is not used. Anyway it has been modified to properly works 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 since they have a new dimension they need to be initialized with a for loop.

- void prvInitialiseGlobalLists(void)

This function is the responsible of initializing the xReadyServersList, pxDelayedServersList 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
This function is the responsible of associate the task in \textit{pxNewTCB} to the server pointed by \textit{pxServer}. Two modifications are made. The first change is relative to the new structure created: \textit{pxTaskArray}. Here is where the new task is registered to the structure, in the form of new array's element. The second change concerns to the behavior of the tasks. In the HSF, in this function tasks are included into 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.

\textbf{Signed portBASE_TYPE prxServerInit(subSCB * pxNewSCB)}

This function is the responsible of registering the server to the scheduler. Furthermore, this function is the responsible of updating the \textit{pxAllServersArray} structure, adding a new element to the array by the function \textit{pvPortRealloc}. As we have assumed, at the beginning all servers are active in all modes, so there is no need to choose where to put the \textit{subSCB}: it must go into the \textit{xServerReadyList}. For future extensions, to choose whether the server is active or inactive is done in this function, by asking to the structure \textit{xServerBehaviorStructure} (already implemented but not used).

\textbf{Signed portBASE_TYPE xIdleServerCreate(void)}

This task was modified to properly set the idle server parameters, using a \textit{for} loop.

\textbf{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 \textit{xServerParameters}. This pointer contains an array of length \textit{N\_MODES}, that is assigned to the \textit{xServerParameterList} field into the \textit{subSCB} structure.

\textbf{Void prvScheduleServers(void)}

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

\textbf{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 functionality of this function is to create a new task in the server. This function also has many changes. Firstly, the parameter \textit{uxPriority} is now a pointer to \textit{portBASE\_TYPE} and contains an array of \textit{N\_MODES} length. And secondly, it has a new parameter added at the end, a pointer to \textit{portBASE\_TYPE} that contains an array again of \textit{N\_MODES} length with the behavior that the task is going to follow. This behavior matrix is the same as used in the \textit{prxRegisterTasktoServer} function, and it determines the manners of the task when a MCR.
arrives. In the function body, there is another modification related with the new way to access to the priority.

- **portTASK_FUNCTION( prvServerIdleTask, pvParameters )**

  The function can also be called an *idle* function. This function has also 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 is following the complete protocol. In the next section (4.3 New API) this procedure is explained in details.

- **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 )**

  This function has the parameters modified in the same way that 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 has been modified to properly add the task to the delayed or overflow queues, having in mind the server's mode.

- **void vTaskDelay( portTickType xTicksToDelay )**

  This function has been modified in the same way that the *vTaskDelayUntil* function.

- **unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask )**

  This function was also modified to give just the task's priority of the current mode of the server.

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

  In the same way that all functions seen until now, this function has been modified from the header to accept a pointer to *portBASE_TYPE* containing an array of *N_MODES* elements. Furthermore, this function has also some modifications when it tries to determine if it could be the next current task.
These functions are the responsible of restore the execution of the system when a suspend function has been called before. These three functions have a modification when they are comparing priorities, due to the data structure changes.

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

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

- void prvSwitchServersOverflowDelayQueue(xList * pxServerList)

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 of accessing the pxReadyTaskList of the server because now it is an array of queues. The second is to release the system to allow the MCR, that it has been blocked in the vTaskIncrementTick, allowing again 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 on 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 this 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 upon the position of the server that is being searched (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 here the time spent 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 have the same pattern as that of 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 it 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 to 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 there is nothing to do with the server’s behavior matrix, so this function is just created like a guideline to 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, puts the field xModeTickCount to zero, deactivates the xCompleteFlag and xSwitchInCourseFlag flags, initializes the xCompleteDelayedTime to zero and it gives to sSwitchModeProtocol the default value of SUSPEND_RESUME_PROTOCOL. This function must be called before any other in the system, because it determines the system’s mode, and there are a lot of things that are depending of that field, such as the server’s
initialization or the task registration. Also this function must not be called twice in the same system execution.

- `void vTaskChangeProtocol(short sNewProtocol)`

  This function is the responsible of changing the protocol for the mode-switch. The different protocols are defined in the `FreeRTOSConfig.h` file and they are the same described in the section 3.3 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`, telling if 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’s why the question “Is a mode-switch in execution” is only asked for that protocol. In the other protocols a task it 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)`

  That function is an easy way to change the protocol, and also to switch the mode. This function 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 turn the flag `uxIsSuspendedFlag` into 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 remove from them; if the task was inactive in the old mode then it updates the `xReadyTime` field and the value of `xGenericListItem`. The update is computing as follows:

  \[
  \text{difference} = xTickCount - xModeTickCount[ \text{auxTSK->sLastActiveMode} ];
  \]

  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 now must 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. Because of this, it is needed to compute a safety margin that it would be two times the server period in the task's last active mode. It means that the task should be executed at least once in the two last server's periods:

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

The variable `savePad` is storing 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 (from now onwards `wakeUpTime`) must have the next time the task must be “awake”, but maybe this time it could happen that was selected for a time where the server's budget is zero, making the task to be awaken after its proper time. If this happen in a normal context there is no problem, because the system can wake up 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) happens then the updated value of `wakeUpTime` (`wakeUpTime'`) it is smaller than the current time (even if we have made a good update) but the task must go to the delayed queue. To avoid a bad allocation of the TCB is necessary to compute a security margin. If the task is executed at least once in two periods of the server then, with the margin computed above is enough to ensure that the task goes into the delayed queue (where it must go).

Briefly, this security margin ensures that the system keep working alright if a mode-switch occurs and a task it was not executed for more time than its own period.

![Figure 10: Usage of the savePad variable.](image)

At this point the task is already located to the right place. It just remains to update the `sLastActiveMode` field and to set `uxIsSuspendedFlag` to false.
void vTaskSwitchMode(short sNewMode)

This function is the responsible of the mode-switch from the current mode (also called old mode) to sNewMode. But 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 to don't disturb during the 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 attend to 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, just to restore the system to the proper task.
- For the abort protocol the only thing that remains is to put all servers remaining-budget to its proper value (the server's budget value), by calling the prvMoveCurrentServerAbortProtocol function, and restore the system.
- For the complete protocol, the procedure is a more complex. At this point of the mode-switch procedure all servers were moved into the new mode queues, except the current server (S0), the one whose task triggered the MCR. The system calls prvMoveCurrentServerCompleteProtocol function which turns on the xCompleteFlag flag and returns pdFALSE and set the field sIncompleteMode to sNewMode. This makes the system to 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 S0 and then executes the next task ready to run. When S0 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 to go directly straight to the prvMoveCurrentServerCompleteProtocol function. This function moves S0 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 \texttt{vTaskSuspendAll}, then the \texttt{xModeTickCount} structure is updated properly the field \texttt{sGlobalCurrentMode} is finally set to \texttt{sNewMode}. Now the interrupts are enabled, \texttt{xSwitchInCourseFlag} is set to \texttt{pdFALSE}, and the tasks are resumed. Then, if it is necessary the system is restored by calling the function \texttt{portYIELD\_WITHIN\_API}, that forces the system to select the proper \texttt{pxCurrentTCB} and execute it suddenly.

- \texttt{void prvMoveCurrentServerAbortProtocol(short sNewMode)}

  The goal of this function is to set all server's remaining-budgets to its 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 \texttt{for} statement is going through the \texttt{xAllServerArray} setting the remaining-budget to the server's budget according to \texttt{sNewMode}.

- \texttt{unsigned portBASE\_TYPE prvMoveCurrentServerCompleteProtocol(short sNewMode)}

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

  The second time this function is called then \texttt{xCompleteFlag} must be set to \texttt{pdTRUE}, which means that the complete-protocol mode-change is ready to finish. Then the function moves the current server to the new mode like \texttt{vTaskSwitchMode} did with the others servers. And now a small trick must be done. The system compute the time spent in complete the server execution and update all task and server different from the idle server and the current server. Finally the function has to set \texttt{xCompleteDelayedTime} to zero, turn off the flag \texttt{xCompleteFlag} and return a \texttt{pdTRUE} value to let the \texttt{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 others tasks to be executed. In this case \texttt{prvMoveCurrentServerCompleteProtocol} moves suddenly 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 others 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 were used to develop HSF. The hardware used is a 32-bit board EVK1100 and the Dragon board as 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, a LCD display. The inside microcontroller is a low-power 32 bits with two memories of 512KB (flash) and 64KB (SRAM). The chips are programmed through the JTAG connector placed on the board.

- To download the code to the EVK1100 board, an AVR Dragon board from Atmel is used. This board is capable of not only to download the code to the microcontroller, but also to debug 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 [9]. This free software is an Eclipse based IDE designed to develop applications over the Atmel devices, supporting a lot 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 classical 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 value: one is debugging the application and suspending it (with a breakpoint or directly with the “suspend” button), and then look for variable and its value. The other way is using the USART through the serial port. But in the machine where the work has been made there is no serial port, so the only way to debug the system it was the first one.
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 have been 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.

<table>
<thead>
<tr>
<th>Priority</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>30</td>
</tr>
<tr>
<td>Budget</td>
<td>15</td>
</tr>
</tbody>
</table>

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 again an empty loop 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.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Task 1 in M0/M1/M2/M3</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3/3/3/3</td>
<td>4</td>
</tr>
<tr>
<td>Period</td>
<td>30/20/35/40</td>
<td>40</td>
</tr>
<tr>
<td>CPU time (in system Ticks)</td>
<td>9/7/6/1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Tasks’ parameter 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. The Figure 12 shows the results for the abort protocol task behavior test.
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 “awaken” it triggers an MCR. This is why all MCR seems to be made before task2 execution, but they are the first thing that task2 perform always.

In the 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 task (Task 1 and Task 2) are executed. Task 2 is responsible of generating the MCR and is executed every 40 system tics. But for example, the third execution of Task 2 is not produced at “80” (when it should be there). 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 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 MCR are grouped into pairs. The first vertical line in the pair represent when the MCR is triggered, the second when it 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 trigger the MCR at the beginning of its execution, and once the task is completed the system switches to task1 without switching the mode. And when task1 is completed and there are no other ready tasks in the server then the system can finally switch completely to “M2”.

**Figure 12: Abort protocol task behavior test result.**
For the rest of the servers and its tasks, the time spent in the execution of the mode-switch is skipped, i.e. all the times are updated and is like there were no transition state. If the current server's remaining-budget is expired during these transition states the system do an exception and let the server to 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 was 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.

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 the server1’s execution, the second the 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

![Figure 13: Result from the complete protocol task behavior test.](image)

<table>
<thead>
<tr>
<th>Server1 in M0/M1</th>
<th>Server2 in M0/M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Priority</td>
</tr>
<tr>
<td>2/2</td>
<td>1/1</td>
</tr>
<tr>
<td>Period</td>
<td>Period</td>
</tr>
<tr>
<td>30/30</td>
<td>34/34</td>
</tr>
<tr>
<td>Budget</td>
<td>Budget</td>
</tr>
<tr>
<td>8/9</td>
<td>15/14</td>
</tr>
</tbody>
</table>

**Table 3: Servers’ parameter 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.

<table>
<thead>
<tr>
<th>Task1 in M0/M1</th>
<th>Task2 in M0/M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Priority</td>
</tr>
<tr>
<td>1/0</td>
<td>4/4</td>
</tr>
<tr>
<td>Period</td>
<td>Period</td>
</tr>
<tr>
<td>30/0</td>
<td>40/40</td>
</tr>
<tr>
<td>CPU time(in system Ticks)</td>
<td>CPU time(in system Ticks)</td>
</tr>
<tr>
<td>9/0</td>
<td>2/2</td>
</tr>
</tbody>
</table>

**Table 4: Tasks’ parameter 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 the server1’s execution, the second the 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 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 execute only (the execution of idle tasks is not shown in the Figures).

Finally, it worth paying attention how this time the period of task2 is respected, executing the MCR every “40” ticks.

![Figure 14: Result from the abort protocol server behavior test.](image)

The second graph shown is from the suspend/resume protocol test, in Figure 15. In this graph it is obvious that the remaining budget is restored from the value it encountered a mode change. When the mode is changed from M0 to M1 for the first time, the server1 had the remaining budget 4, and server 2 has the remaining budget 10. When the system changes to the M0 at tick 80, the server 1 and server 2 start their executions with the remaining budgets 4 and 10 respectively. Due to the method used to measure the server’s remaining-budgets in Figure 15 the graphic shows the remaining-budget in 9. This is because the measures are taken in every system tick, before the system decrements the remaining-budget. So, talking properly, when the MCR is triggered at 80, the remaining-budget of server2 is 9, and then in the next MCR the system is restoring the remaining-budget to 9.

![Figure 15: Result from the suspend/resume protocol server behavior test.](image)

This behavior is easy to understand watching the server2 execution, in the second mode-switch to “M1”. There is also a singular behavior in the last mode-switch. In all the figures task2 is executed after the mode-switch, but here is executed before and after the MCR. This is because in the previous execution of task2 it does not complete the task, but it has to be preempted because the server’s remaining-budget expires, then, when the server is ready again, the task recover its last state, complete the last execution and then go to sleep until the next period reach. But “the next period” had already come while the server was in the release-queue,
so the system executes again task2: first the MCR and then the CPU consumption time, as always did. Also is remarkable how server1 restore its remaining-budget after every MCR.

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 a step in the scope of server2’s remaining-budget. Theoretically this scope should not have these horizontals steps. As clear in Figure 13, the MCR are represented now by a pair of verticals lines, the first is when the MCR is triggered and the second 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: Result 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. For every test it will be a prior explanation with the main features of the test for the better understanding and the results are discussed in the next section 5.4 Discussion. All measures shown are in microseconds (us), and the resolution of the system is 10 us. In the entire test the system goes through four modes, following the progression: $m_0-m_1-m_2-m_3-m_0-...$. The server parameters are not significant for this test because it does not affect the mode-switch behavior, just for the complete protocol. But the time values are depending on what is the CPU load of the task executed within the transition. With respect to the complete protocol time values, there are two columns, the first show the time spent in the second call of the $vTaskSwitchMode$ function, i.e. the time spent in changing the current server mode and to update 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 last 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.

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

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 number of tasks in a server.

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

Table 6: Time values for the second time test.

III) The third test consists of two servers with one task each.
IV) The fourth test consists of one server with four tasks.

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

Table 7: Time values for the third time test.

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

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

Table 8: Time values for the fourth time test.

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

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

Table 9: Time values for the fifth time test.

VII) The last test performed tries to imitate a real scenario, where the task has different behaviors and different executions time 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 tasks behaviors in Table 12. The tasks periods and CPU times have no interest, just know that in mo the task that trigger the MCR spends a lot of CPU time after the mode-switch request. That is made to simulate a real scenario where the task responsible of trigger the MCR has also some other things to do.

<table>
<thead>
<tr>
<th></th>
<th>Server1</th>
<th>Server2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 11: Server parameters.
<table>
<thead>
<tr>
<th>Period</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 11: Servers parameters for the real scenario test.

<table>
<thead>
<tr>
<th>Task1</th>
<th>Task2</th>
<th>Task3</th>
<th>Task4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>Inactive</td>
<td>Active</td>
<td>Active</td>
</tr>
<tr>
<td>M1</td>
<td>Active</td>
<td>Inactive</td>
<td>Active</td>
</tr>
<tr>
<td>M2</td>
<td>Active</td>
<td>Inactive</td>
<td>Active</td>
</tr>
<tr>
<td>M3</td>
<td>Active</td>
<td>Active</td>
<td>Inactive</td>
</tr>
</tbody>
</table>

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 there are the values obtained in the first four mode-switches.

<table>
<thead>
<tr>
<th></th>
<th>Abort</th>
<th>Suspend/Resume</th>
<th>Complete</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Average</td>
<td>311,390</td>
<td>285,250</td>
<td>192,550</td>
<td>2036,180</td>
</tr>
<tr>
<td>St. deviation</td>
<td>11,721</td>
<td>19,034</td>
<td>10,436</td>
<td>3004,754</td>
</tr>
<tr>
<td>Max Time</td>
<td>330</td>
<td>309</td>
<td>202</td>
<td>9610</td>
</tr>
<tr>
<td>Min Time</td>
<td>298</td>
<td>256</td>
<td>170</td>
<td>437</td>
</tr>
<tr>
<td>Values (us):</td>
<td>298</td>
<td>256</td>
<td>181</td>
<td>9578</td>
</tr>
<tr>
<td></td>
<td>298</td>
<td>309</td>
<td>170</td>
<td>437</td>
</tr>
<tr>
<td></td>
<td>330</td>
<td>288</td>
<td>192</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td>309</td>
<td>288</td>
<td>192</td>
<td>458</td>
</tr>
<tr>
<td></td>
<td>309</td>
<td>256</td>
<td>192</td>
<td>5642</td>
</tr>
</tbody>
</table>

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 $M_0$ to $M_1$ due to the big CPU load of the current server at the MCR moment.
5.4 Discussion

About the behavior test there is not so much to discuss. The graphics reflect that the system behaves as it is expected. Maybe there is a point where the system does not seem to behave perfectly. Watching the servers’ behavior test, specifically from the abort protocol results it can be seen how, after the MCR, the remaining budget of server2 decreases a little bit. Ideally, after the transition under the abort protocol, the highest priority ready server must be executed, that means server1 but instead is executed server2, leading to explain the two different kinds of mode-switch behaviours that are presented for the abort protocol. The first one has been used to do the behavior test. Once the system has completed the MCR and if a task is still active, continues the server to be executed until the next system tick. That procedure is called the soft ending. The second is called hard ending which 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 into the highest priority ready server and set it as the current task. Which procedure to be used can be configured in the FreeRTOSConfig.h file by giving value “0” for soft ending or “1” for 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 time. So this leads to a dilemma: the hard ending procedure is expensive in time but follows the ideal behavior while the soft ending procedure is better in performance but does not follow the right behavior.

Then, which procedure should be chosen?

This question does not have a correct answer; the choice belongs to the users. But there are some factors that point to the soft ending as the better choice. To prove it, the attention must be focus now on the performance results. The first test just measures the performance of an empty system and the aim of this measure is to take it as a reference to know how the performance varies with servers or tasks. More interesting results are 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 does not reach during the execution of the mode-switch (i.e. no system tick it has been missed) probably it is close to reach. Then, all mode-switch 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 it has the highest priority or not. If the hard ending is being used probably the system will 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 will execute a schedule function again.
- If no tick was missed and the task does not has the highest priority, the system will schedule to another task, but not for a entire tick.
- If no tick was missed and the task is the most priority then the system will keep executing it.

Then, the only case in which a hard ending is useful is the third case. And even if the reschedule is justified like in the third case the executing time until the next tick it could be,
depending on the tasks and servers amount, very short.

Now the discussion will 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. And 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.

About the complete protocol times there is nothing interesting to say. Paying attention to the second column (the one that represents the time spent in the whole transition) it can be appreciated 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 been respectful with the preceding code, trying to modify it as least as possible, easy to configure and discard if it is necessary and fully commented to help future users or developers.

Finally some additions to the explained code has 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 possible to share resources between modes.
− The capability of declaring new modes during run time.

It is proposed to improve the actual system for a better performance:

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

Finally it is proposed to do a complete schedulability analysis.
7. References


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 it can be found in section 4.3.