Mode-Change Mechanisms support for Hierarchical FreeRTOS Implementation

Rafia Inam*, Mikael Sjödin*, Reinder J. Bril†
* Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
† Technische Universiteit Eindhoven (TU/e), Eindhoven, The Netherlands
rafia.inam@mdh.se, mikael.sjodin@mdh.se, r.j.bril@tue.nl

***Abstract*—Multi-mode embedded real-time systems exhibit a specific behaviour for each mode, and upon a mode-change request the task-set and timing interfaces of the system need to be changed. This paper presents the implementation of a Multi-Mode Adaptive Hierarchical Scheduling Framework (MMAHSF) and provides a generic skeleton (framework) for a two-level adaptive hierarchical scheduling supporting multiple modes and multiple mode-change mechanisms on an open source real-time operating system (FreeRTOS). The MMAHSF enable application-specific implementations of mode-change protocols using a set of predefined mode-change mechanisms.**

The paper addresses different mode-change mechanisms at both global and local scheduling levels. It presents examples of mode-change protocols that are developed by composing together these mechanisms in multiple ways and provide the initial results of executing these protocols in the MMAHSF implementation on an AVR 32-bit board EVK1100.

***Keywords*-real-time systems; hierarchical scheduling; multi-mode systems; mode change protocols**

I. Introduction

Real-time embedded systems often have to support very different and changing application scenarios. A *multi-mode system (MMS)* is said to operate in multiple *system modes*, where each mode corresponds to a specific application scenario. A mode is typically represented as a unique set of tasks with its specific functional and non-functional properties. The system changes from one mode to another upon some condition at runtime, and at a specific time it can be in one of the predefined modes. The transition from one mode to another is done using a *mode-change protocol*. The mode-change protocol will, in turn, use a set of *mode-change mechanisms* to effectuate the mode-change. In this paper we present a set of mode-change mechanisms that can be composed to application-specific mode-change protocols. We also demonstrate for some examples how such protocols are constructed using the mechanisms.

Our work is performed in the context of the hierarchical scheduling framework (HSF) [1], which is a known technique used to partition a system into a set of subsystems, each consisting of its own set of tasks and to provide temporal isolation by executing multiple applications in the subsystems. In the HSF, the servers' reservation parameters may need to change from mode to mode at runtime. Numerous studies are found on adaptive reservation techniques for server-based multi-mode systems from a scientific perspective [2], [3], [4], [5], however to the best of our knowledge, no work is done from an implementation perspective. We focus on implementing the resource reservations for a two-level hierarchical scheduling to adapt modes at runtime and call it a *multi-mode adaptive hierarchical scheduling framework (MMAHSF).*

An application may not only need to execute in multiple modes but also require multiple mode change protocols to respond to changes in the environment. For example, consider an application that can be in normal execution, emergency, and shutdown modes. The mode change during *normal execution* can suspend old mode's tasks execution and resume tasks of the new mode. Sometimes it is required to complete the execution of some important tasks before changing the mode during the normal situation. 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* situation may require aborting all the tasks instantly. Similarly consider the example of downloading a file, where downloading *suspends* for some time due to the decrease in bandwidth. Later the user likes to *resume* downloading process from where it stopped earlier rather than restarting it. In some other situation, an application may need to *reset* itself by restarting all of its tasks. To address this issue, we implement a generic MMAHSF that can support multiple mode-change semantics for a hierarchical system along with multiple operational modes. The mode-change mechanisms can be combined at the server and task levels in multiple ways to build different mode change-protocols.

Contributions: The main contributions are as follows:

- We provide *multi-mode system support for two-level hierarchical scheduling* using the FreeRTOS operating system. Our HSF implementation is based on idling periodic servers, using fixed-priority preemptive scheduling at both (global and local) levels of hierarchy. For the extension with multi-mode support, our main considerations are to minimize the changes in the FreeRTOS kernel and to keep the original API semantics.
- We illustrate the *multi-mode semantics for three mode-change mechanisms, i.e. suspend-resume, complete, and abort mechanisms*, and describe their practical implementation issues.
- We apply these mode change mechanisms to *develop different mode-change protocol examples*; e.g. suspend-resume protocol, reset protocol, etc.

- We *illustrate the resulting behavior* of mode-change protocols through a set of experiments. The experiments are performed on an AVR 32-bit board EVK1100 [6]. We also measure the overheads of some protocols.

The MMAHSF discussed in this paper is an extension of our previous work [7]: both papers are based on the idea of a generic framework for a multi-mode system using a hierarchical system. The work described in this paper is the actual implementation of the framework. The previous work, on the other hand, focused on just the presentation of the general idea of MMAHSF at a high level of abstraction and described system's main goals and challenges. It lacked the mode-change mechanisms, and a practical implementation and the behaviour evaluation of the system. Another significant extension in this paper is the description and implementation of different mode-change protocols. Further, we also provide the results of our evaluations of the MMASHF implementation using different protocols.

Paper Outline: Section II provides an overview of the HSF and its implementation on FreeRTOS that our work uses and presents the related work on multi-mode systems. Section III describes the support for MMS inclusion into the HSF implementation, system model and assumptions. It also explains the system's design and functionality. We explain the semantics of three mode-change mechanisms in section IV. In section V we present exemplar mode-change protocols that are developed using these mechanisms. In section VI we explain some important implementation details of MMAHSF. We develop different mode-change protocols, test the system behaviour in different modes, and measure the execution overhead of these protocols. In section VII we conclude the paper with a description of future work. We provide the API of our implementation in the Appendix.

II. Background and Related Work

This section presents an overview of the Hierarchical Scheduling Framework implementation in FreeRTOS, followed by related work on multi-mode systems.

A. HSF and its implementation

Our implementation of MMS is based on a two-level HSF implementation for the FreeRTOS operating system [8] that follows the periodic resource model [9]. In a two-level HSF, the CPU time is partitioned among many subsystems (or servers), that are scheduled by a global (system-level) scheduler. Each server contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler as depicted in Figure 2. FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system that is ported to 23 hardware architectures ranging from 8-bit to 32-bit micro-controllers, and supports many development tools [10]. Its main advantages are portability, scalability and simplicity. The core kernel is simple and small, with a binary image between 4 to 9KB. Since most of the source code is in C language, it is readable, portable, and easily expandable and maintainable.

The official release of FreeRTOS only supports single level fixed-priority preemptive scheduling. However, we have implemented a two-level hierarchical scheduling framework for FreeRTOS [8] with associated primitives for hard real-time sharing of resources both within and between servers [11]. The HSF implementation uses fixed-priority preemptive scheduling policy at both global and local levels for two kinds of servers: idling periodic [12]; and deferrable servers [13]. The Stack Resource Policy (SRP) [14] is implemented for local resource-sharing (within a server), and the Hierarchical Stack Resource Policy (HSRP) protocol [15] is implemented for global resource-sharing (between servers) with three different overrun mechanisms (without payback, with payback, and enhanced overrun) to deal with the server budget expiration within the critical section [16]. The HSF supports reservations by associating a tuple $\langle Q, P \rangle$ to each server where P is the server period and Q ($0 < Q \leq P$) is the allocated portion of P. Given Q, P, and information on resource holding times, the schedulability of a server and/or a whole system can be calculated with the methods presented in [9] and the overheads of servers execution are added in [11]. The implementation has been tested and experimental evaluations have been performed on a 32-bit AVR-based micro-controller board EVK1100 [6].

B. Multi-Mode systems

The scheduling theory for multi-mode real-time systems and for component-based multi-mode systems has been intensively investigated. Sha *et al.* [17] provided a simple mode-change protocol for a prioritized preemptive scheduling environment. A survey on mode-change protocols for fixed-priority preemptive scheduling (FPPS) using a single processor is presented in [18] and along with proposed several new protocols. Mode switch problem for dynamic scheduling using Earliest Deadline First (EDF) is considered in [19], [20]. Multi-mode real-time schedulability analysis for different assumptions and models is presented in [21], [22], [4], and added to the compositional system using Real-Time Calculus (RTC) in [3]. Some frameworks and programming languages support multi-mode systems, including [23], [24] and [25], [26], [27] respectively. Hang *et al.* [28] provides the details of mode switch logic algorithms to handle mode mapping for component-based systems.

Static resource reservations for servers [29], [12], [13] are not suitable for multi-mode server-based systems where resource reservations vary with the change of mode. Hence reconfigurable (adaptive) servers are suggested for dynamic reservations by Abeni *et al.* [30], [31], where [31] provides adaptive reservations using feedback scheduling. Dynamic reconfiguration of servers for multi-mode system is addressed in [2], [32]. Stoimenov *et al.* [32] provides guaranteed resource provisioning during mode changes by using TDMA servers. Santinelli *et al.* [2] addresses the problem of timing analysis during the reconfiguration process.

A mode-change protocol is implemented for reallocating the memory among tasks in [33] but no work has been done to reallocate the CPU time. To the best of our knowledge, no work has been found with respect to the MMS implementation using hierarchical scheduling.

III. Multi-Mode Adaptive Hierarchical Scheduling Framework

This section describes a generic framework for the implementation of multi-mode system for hierarchical scheduling. The purpose is to develop a basic skeleton that is capable of incorporating multiple mode-change protocols to change the system mode. This section starts with the introduction of the multi-mode adaptive hierarchical scheduling framework followed by the system model, basic assumptions of the implementation, design details and the system's functionality.

HSF becomes adaptive in nature by incorporating different modes of the system and using a mode change protocol to change the system-mode at run-time. Normally, a different piece of software is executed for each mode, i.e. a different task set, implementing a different functional and non-functional characteristics, is executed. As a consequence of a changed task set execution within a the server for each mode, the server's timing interfaces are modified for each mode. The system can be viewed as a set of hierarchically organized subsystems, each subsystem consisting of its own set of tasks.

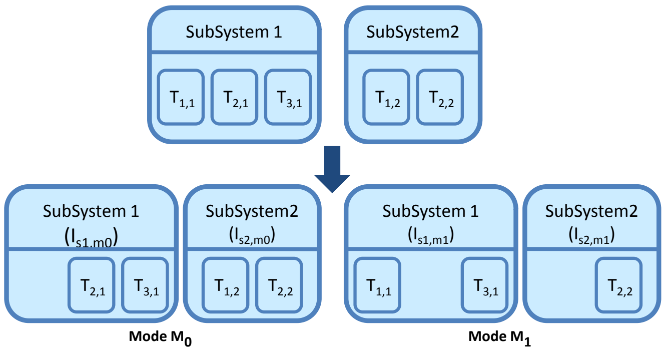


Fig. 1. An example of a Multi-Mode Hierarchical System

For example in Figure 1, a multi-mode hierarchical system consists of two subsystems S_1 and S_2; which in tern consists of task sets $\mathcal{T}_1 = \{\tau_{1,1}, \tau_{2,1}, \tau_{3,1}\}$, and $\mathcal{T}_2 = \{\tau_{1,2}, \tau_{2,2}\}$ respectively. The system supports two different modes M_0 and M_1, both having different timing properties. Both subsystems are active in both modes while the tasks can be active or inactive in a particular mode. The task $\tau_{1,1}$ is inactive in mode M_0, tasks $\tau_{2,1}$ and $\tau_{1,2}$ are inactive in M_1, while tasks $\tau_{3,1}$ and $\tau_{2,2}$ are active in both modes. In the rest of this paper, we use the term subsystem and server interchangeably.

A. Terminology

The following terms are used in this paper:

- **Active/Inactive tasks:** Those tasks that belong to the currently executing mode are called *active*, while the tasks that do not belong to the currently executing mode are called *inactive* or *deactivated* tasks. The deactivated tasks can start execution only when their mode starts execution later.
- **Old/New mode:** Upon a mode change request, the system switches from the currently executed or *old* mode to the newly requested or *new* mode.
- **Steady state:** System executing within an individual mode is said to be in a steady or stable state, e.g. the system executing in the old mode or in the new mode is in the steady state.
- **Transition state:** The time interval $[t_{req}, t_{end}]$ during which a mode-change protocol executes to change system mode from old to new. t_{req} denotes the time instant at which the mode change is requested, and t_{end} is the time instant at which the mode change is completed and the system starts it execution in the new mode. The servers change their task sets and interfaces during this interval, and the overall system is neither in the old nor in the new mode.
- **Mode change requesting task:** A mode change request is raised by an executing task called the mode change requesting task.
- **Mode change requesting server:** A server during whose execution a mode change request is raised is called a mode change requesting server. The resource provisioning of the server is aborted any time during the mode transition interval $[t_{req}, t_{end}]$ depending on the mode change protocol.

B. Mode Change Request Controller

The mode change protocol is performed within a *Mode Change Request Controller (MCRC)*. The paper addresses system modes and system-mode changes, hence to change the mode of the whole system in hierarchical scheduling, the mode-change has to be done at both global and local levels. Therefore, we use a *global MCRC* and a *local MCRC* as shown in Figure 2. Any mode-change involving only a (single) local mode-change is therefore out-of-scope. The global MCRC is responsible for changing the mode of the whole system upon a mode-change request, and calls the local MCRCs to change the mode of each subsystem (i.e. change the task set). The local MCRC is responsible for handling the mode-change locally; hence change the mode of the subsystem.

Figure 3 depicts a mode change for the system described in Figure 1. The system mode is changed from the old mode M_0 to the new mode M_1, each exhibiting a unique behaviour. A mode change happens during the mode transition interval $[t_{req}, t_{end}]$. It is shown in Figure 3 that each subsystem exhibits a unique timing interface for each mode and during the transition state these interfaces are changed. A particular mode change protocol is executed during the transition state depending on the system's changing conditions.

C. System model

We consider a two-level MMAHSF, in which a global scheduler schedules a system that consists of a set of subsystems $\mathcal{S}$ and a set of modes $\mathcal{M} = \{M_0, \ldots, M_{n-1}\}$ where n is the total number of modes in the system, and

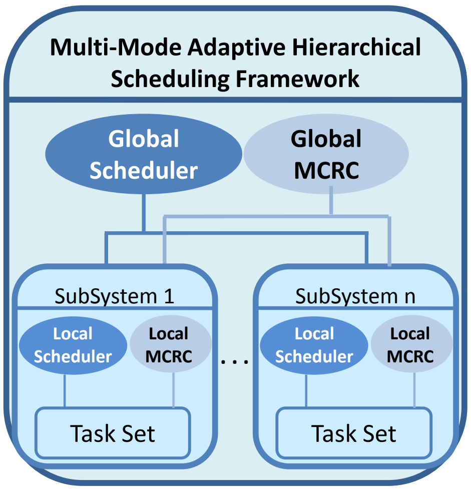


Fig. 2. Multi-mode adaptive hierarchical scheduling framework

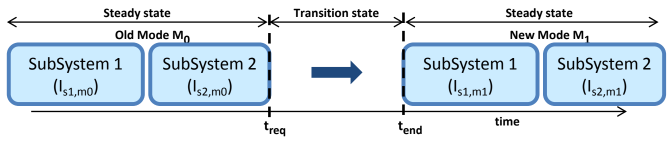


Fig. 3. A mode change from the old- to the new-mode

a global MCRC to change the system's mode. Each subsystem S_s consists of a local scheduler along with a set of tasks $\mathcal{T}_s$ and a local MCRC. Fixed-priority preemptive scheduling is used at both levels of schedulers. For each mode M_m, each subsystem S_s is specified by a different *timing interface* $I_{s,m} = \langle P_{s,m}, Q_{s,m}, p_{s,m} \rangle$ and a subset of tasks $\mathcal{T}_{s,m} = \{\tau_{1,s,m}, \ldots, \tau_{n,s,m}\}$, where $P_{s,m}$ is the period for that subsystem ($P_{s,m} > 0$), $Q_{s,m}$ is the capacity allocated periodically to the subsystem ($0 < Q_{s,m} \leq P_{s,m}$), a unique priority $p_{s,m}$ in mode M_m, and $\mathcal{T}_{s,m} \subseteq \mathcal{T}_s$.

MMAHSF contains a set of mode change mechanisms $\mathcal{C} = \{C_0, \ldots, C_{i-1}\}$ where i is the total number of mode change mechanisms implemented in the system.

D. Assumptions

The following assumptions are made for the initial design:

- The set of subsystems (S) is fixed in the system. A subsystem without any active task (in a particular mode) can be considered as inactive and the interface of an inactive subsystem is set as $Q_{s,m} = 0$. Similarly, a subsystem with at least one active task is considered active.
- The number of modes ($\mathcal{M}$) is fixed in the system and new modes are not allowed to be added at run-time. The interfaces of all subsystems for all modes are defined statically.
- Tasks can be active or inactive in a mode. The timing behaviour of a task remains the same in all modes. $\mathcal{T}_s$ is fixed hence dynamic creation and destruction of tasks are not allowed.
- Only intra-subsystem intra-mode resource sharing is assumed. It means that resources that are shared among the tasks of the same subsystem are mode-specific; e.g. resources shared in mode M_1 are not shared with the tasks of mode M_2.
- The interrupt handlers are mode specific and the interrupt handling only works for the current mode. Interrupts that have no attached handlers (e.g. during mode changes) are ignored.

E. The design of the scheduling hierarchy

To get minimal changes and better utilization of the system, we have matched the design of the MMAHSF implementation with the underlying FreeRTOS operating system. This includes consistency from the naming conventions to API, data structures and coding style. It will also increase the usability and understandability of our implementation for FreeRTOS users.

Each subsystem S_s is reflected by its interface. The server type is idling periodic here and we are using fixed-priority preemptive scheduling with the FIFO (to break ties between equal priorities) at both levels. The system stores the `current mode` of the system, and a total `number of modes` i.e. n in the system. During execution, the system can be in one of these predefined modes $M_0, \ldots, M_{n-1}$. We also store `current mode change protocol`, and a total `number of mode change protocols`.

For each server, the interface $I_{s,m}$ and lists for tasks $\mathcal{T}$ of each mode M_m are stored in a subsystem control block `subSCB`. Since a server may have different timing interfaces (periods, budgets, and priorities) in different modes, these three properties are stored in an array in the system. Each server also maintains a currently running task and two lists to schedule its tasks: a ready-task list, and a delayed-task list for each mode. Some tasks of a server will be active in one mode but inactive in another mode. The deactivated tasks of a server in a mode should not interfere with the execution of active tasks of the server in another mode for better system performance. Therefore, separate ready-task and delayed-task lists per mode are used to keep track of ready and delayed tasks respectively. Moreover during system execution, only the lists of the currently executing mode will be active in the system, and the lists of all the other modes are inactive (means not accessed). Since the total number of modes are fixed in the system, lists for each mode are stored within an array, where the array index specifies a unique mode. A detailed design with diagrams and design considerations is presented in [7].

F. System functionality

An event-triggered mode-change request initiates the process of mode-change. The request could be triggered either internally (e.g. deadline misses of tasks or too long execution of the idle task) or by external user request (we assume that the triggers are handled by tasks). Global MCRC is called upon a mode-change request to change the system's mode.

The global MCRC is called from the system tick-handler upon a mode-change request. It executes a mode-change mechanism and changes the old-mode to the new-mode using the current mode change protocol. It changes the timing interfaces for all the servers from the old-mode to the new-mode. The local MCRC is responsible for activating the lists belonging to the new-mode and for deactivating the lists belonging the old-mode. The active lists of the servers only include all those tasks that belong to the new-mode of the system. Tasks that belong (or are active) in both the old and new mode are copied to lists of the new mode. In the end, the system's `current mode` is changed to the new-mode.

IV. Mode-Change Mechanisms

The system is able to switch from one mode to another by making a Mode-Change Request (MCR). The MCR initiates the process of mode-change using the API `vTaskSwitchMode(sNewMode)`, by changing the system's current (old) mode to the new mode.

The transition from the old to the new mode is handled by the mode-change protocol executed during the transition state. A mode change request is raised by a currently executing task of the currently executing (mode changing) subsystem which aborts the resource provisioning of its server any time during the transition interval $[t_{req}, t_{end}]$. All other non-executing subsystems will change instantaneously to the new mode.

We have implemented three different basic mode change mechanisms at both global (servers) and local (tasks) levels which are subsequently applied to develop different mode change protocols. These mechanisms are Suspend-Resume mechanism (SRM); Complete mechanism (CM); and Abort mechanism (AM). The servers and tasks behaviour during the transition state and in the new mode depends on the mode change mechanisms.

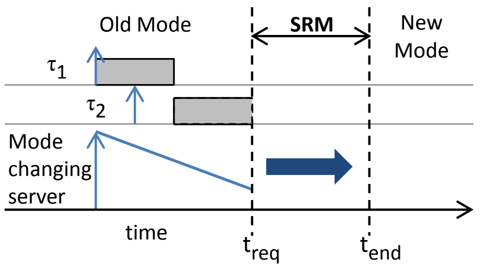
The three mechanisms and their effect on the execution of servers and their tasks are described in this section. Mode-change mechanisms for local and global levels during the transition state are depicted in Figure 4.

A. *Suspend-Resume mechanism (SRM)*

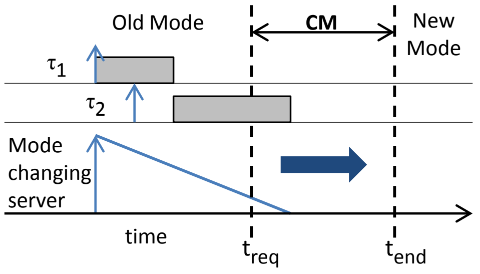
For a suspend-resume mechanism, the state of servers (running, ready, or suspended), the state of tasks (running, ready, waiting or suspended), and the structures of servers and tasks (data, objects and resources, server control block, and task control block) are stored when they are suspended. Later they are resumed from the same stored point.

1) At the global level: The resource provisioning of servers is stopped as shown in Figure 4(a). The timing interface of servers including their remaining budget of the server for the old mode is stored in the system at the time of suspension. In the new mode, servers are resumed from their stores states with the remaining budget at which they were previously suspended in this mode.

2) At the local level: All tasks of the new mode are suspended and their states are stored, e.g, task τ_2 is suspended in Figure 4(a). It is possible since separate ready-task and delayed-task lists per mode are used to store tasks' states per mode. All suspended tasks (of the new mode) are resumed



(a) Transition using Suspend-Resume mechanism



(b) Transition using Complete mechanism

Fig. 4. Mode-change mechanisms at local and global levels.

from their stored states. All the events that occurred during the deactivated state of the tasks are delayed to be handled until their corresponding mode become active and start execution.

B. *Complete mechanism (CM)*

For a CM at the global level, the currently executing server (of old mode) keeps on providing its service. At the local levels the old-mode tasks in ready queue are allowed to execute their current activations during the transition state until either these tasks are completed or the server is depleted. As shown in Figure 4(b) task τ_2 is allowed to complete. The replenishment of server budget is not allowed during the transition state.

In the new mode at the global level, the server replenishes with the full budget. At the local level, if tasks have completed their execution then they are restarted, otherwise in case of server depletion they are resumed in the new mode.

C. *Abort mechanism (AM)*

For AM, the execution of servers at global level and their task sets at local level are aborted and their mode is changed immediately. Note that the servers and tasks are not suspended as in SRM; rather they are aborted, which means that their internal states are not stored in the system. In the new mode, the servers and tasks are restarted at global and local levels respectively.

V. Exemplar Protocols

Using the composition of mode-change semantics for servers and tasks, different mode-change protocols can be developed and applied to a system. Different subsystems may have different mechanisms, e.g. one subsystem could use CM while others could use SRM etc. For these protocols, the mechanisms used for a server and tasks of the server remain the same. In this sections we present exemplar mode-change protocols in a hierarchical mode-changing system. The behaviour of servers in the new mode using these protocols are depicted in Figure 6.

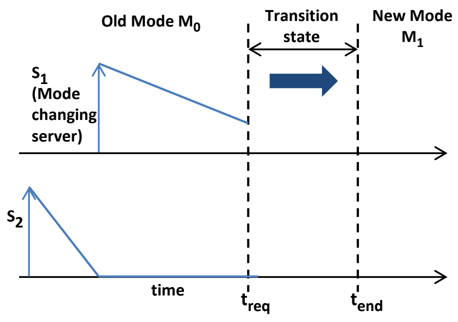


Fig. 5. Servers before the mode change request

Figure 5 presents a hierarchical system before a mode change. The system consists of two servers S_1 and S_2 where S_1 has a higher priority than S_2. For simplicity we are not displaying tasks here. The system is in mode M_0 and at time t_{req} a mode change request is made. At t_{req}, the server S_1 is executing, while S_2 has expired its budget and is waiting for its next activation.

The tasks can be either active or inactive in the new mode; therefore, their behaviour in the new mode depends upon the mode-changing mechanism executed during the transition state and on whether they are active or inactive in the new mode. The behaviour of tasks is described in the exemplar protocols.

A. Suspend-Resume Protocol using SRM

Using suspend-resume protocol, all old-mode (executing and non-executing) servers and their tasks are suspended (deactivated), and all new-mode servers and tasks are resumed (activated). We depict an example of *suspend-resume protocol* using SRM at the global level in Figure 6(a). For better understanding we consider that the system presented in Figure 5 is suspended during the transition state using SRM for both servers and their tasks. The system is resumed to mode M_0 again later in time as depicted in Figure 6(a) where both servers S_1 and S_2 are resumed. S_1 starts its execution according to its saved remaining budget of mode M_1. The behaviour of tasks is summarized as follows:

- Tasks that are inactive in the new mode are suspended during the transition state.
- Tasks that are active in the new mode are resumed during the transition state and will execute in the new mode.
- Tasks that are active in both the old and new modes will remain active in the new mode. They are copied to the lists of the new mode. The currently executing and ready tasks of the old mode will be copied to the ready list of the new mode, while the delayed tasks of the old mode will be copied to the delayed list of the new mode. The currently executing task is allowed to continue its execution in the new mode according to the priorities of the ready tasks of the new mode. The list items are pointers, so copying the pointers is efficient.

B. Reset Protocol using AM

Similarly, a *reset* or *abort protocol* is developed using AM at both levels for all servers and for all tasks in the system. The system in Figure 5 is aborted during the transition state using AM and after the transition at time $[t_{end}]$ the whole system (including its servers and tasks) is restarted as described in Figure 6(b). Since the servers and tasks are aborted, therefore, in the new mode, all servers are replenished with their full budgets and periods, and their tasks are restarted (not resumed as in other protocols). The behaviour of tasks is as follows:

- All tasks regardless they are active or inactive in the new mode are aborted during the transition state and moved to the ready lists.
- All tasks that are active in the new mode will be restarted in the new mode.

C. A Protocol using a combination of CM and SRM

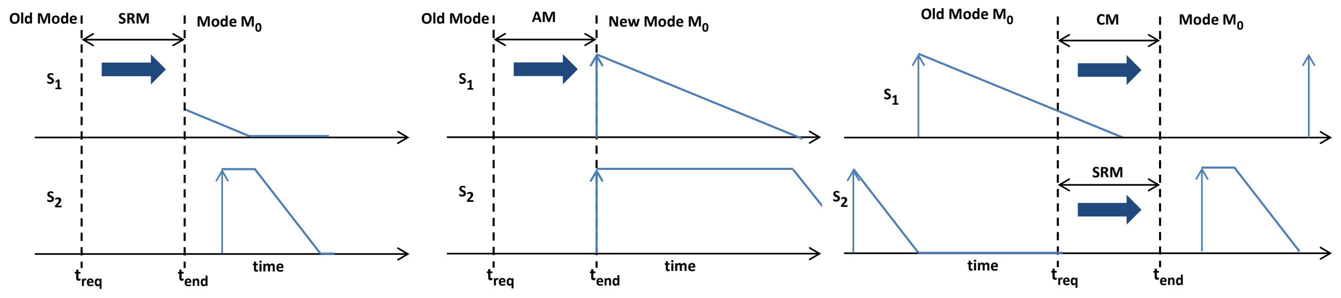
A complete protocol can be made using CM during the transition state where server and its tasks have completed their execution. At the global level, the server is replenished with its full budget in the new mode. Similarly at the local level, since tasks have completed their execution, they are restarted in the new mode. We present an example in Figure 6(c) where the server S_1 (including its tasks) is executing CM while server S_2 and its tasks execute SRM. In the new mode, S_1 replenishes with its full budget and its tasks restart their execution, while S_2 and its tasks are resumed.

VI. Implementation and Experimental Results

The multi-mode hierarchical system can be executed by setting a macro `configMULTI_MODE` as 1 in the configuration file `FreeRTOSConfig.h` of the FreeRTOS. Since we have kept the original FreeRTOS API intact and the design of MMAHSF implementation is compatible with the underlying operating system, it is possible to execute the original FreeRTOS scheduler by setting the macro to 0. The total number of modes `N_MODES` in the system, an initial executing mode, and mode-change mechanisms are defined in the configuration file. It provides a freedom to the developer/user to create and add new mode-change mechanisms in the system. An initial mode-change mechanism can also be set (by default suspend-resume is used for all servers and tasks in the system).

The `xServerCreate()` and `xServerTaskGenericCreate()` APIs create server and tasks in a server respectively, and are modified to incorporate multiple modes and the timing parameters for each mode for servers and tasks respectively. `vTaskStartModeScheduler(defaultMode)` API initializes all variables and fields related to modes. It initializes the system to the initial mode and the initial mode-change mechanism for both global and local levels. A detailed list of all modified and new APIs and macros are provided in the Appendix.

The scheduler is started by calling `vTaskStartScheduler()` (typically at the end of the `main()` function), which



(a) Suspend-Resume protocol at global level (b) Reset protocol at global level (c) A Protocol using combination of CM and SRM

Fig. 6. The semantics of servers using different mode-change protocols.

is a non-returning function. Depending on the value of the `configMULTI_MODE` macro, either the original FreeRTOS scheduler or MMAHSF scheduler will start execution. `vTaskStartScheduler()` initializes the system-time to 0 by setting up the timer in hardware.

The mode-change mechanism of the system can be changed at runtime by calling `vTaskChangeProtocol(sNewProtocol)` API. As described earlier, a task can be active or inactive in a particular mode and it is set in the `xBehaviorTaskMatrix` within the `tskTCB` structure when the task is created. `xTaskChangeTaskModeBehavior(mode, xBehavior)` API modifies the task's behaviour (i.e. active or inactive) in a particular mode.

The system's mode is changed from current mode to the new mode using `vTaskSwitchMode(sNewMode)` API. The system first changes the mode of servers at the global level by removing servers from old mode's queue using `vTaskChangeServerModeBehavior(mode, xBehavior)` API, changes the `sCurrentMode` value to the `sNewMode`, and finally reallocates servers to the new mode's queues depending in the mode-change mechanism. For AM, all servers remaining budget is set to the full budget and are moved to the ready queue of the new mode. For SRM, all servers with a remaining budget greater than zero are moved to the ready queue while other servers are moved to the delay queue of the new mode. For CM, `prvMoveCurrentServerCompleteProtocol()` function allows the execution of tasks until completion or until server depletion. The server is then moved to the delay queue of the new mode. Similarly, at the local level, `prvMoveTaskToNewMode()` function is called from within the `vTaskSwitchMode()` and moves all tasks of the server to the task queues of the new mode.

A. Experimental setup

Now we present the experiments to test the behaviour and performance of our MMAHSF implementation. For this purpose, we develop some examples of mode-change protocols using different mode-change mechanisms for tasks and servers. The examples are executed and all measurements are performed on the target platform EVK1100 [6]. The `AVR32UC3A0512` micro-controller runs at the frequency of `12MHz` and its tick interrupt handler at `1ms`.

B. Behavior testing

Here we illustrate and evaluate mode-change protocols by means of specific combinations of mode-change mechanisms for servers and tasks. This test is performed to observe the behavior of servers in different modes using different mode-change protocols.

We perform three experiments to test the behavior of our MMAHSF implementation by executing idling periodic server and demonstrate our results by means of a trace of the execution. Two servers S_1 and S_2 are used in the system having two modes M_0 and M_1, where S_1 has higher priority than S_2. The servers in both modes used to test the system are given in Table I.

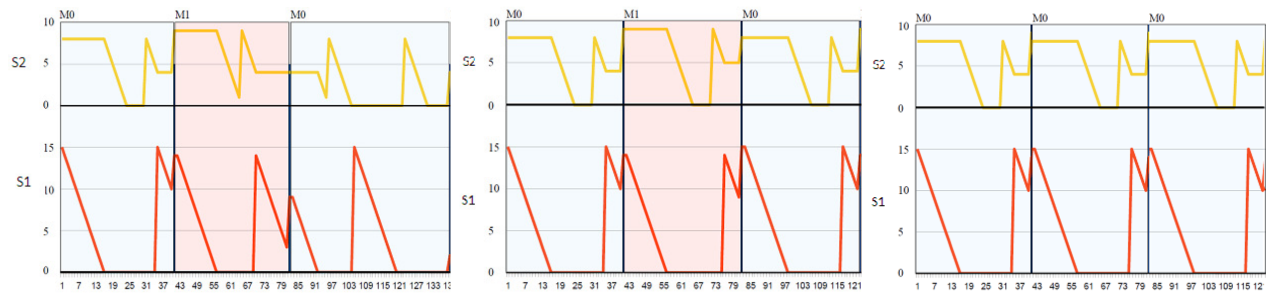
Servers	S_1		S_2	
Modes	M_0	M_1	M_0	M_1
Priority	2	2	1	1
Period	34	34	30	30
Budget	15	14	8	9

TABLE I
SERVERS INTERFACE TO TEST SYSTEM BEHAVIOUR.

Servers	S_2		S_1	
Modes	M_0	M_1	M_0	M_1
Tasks	$\tau_{1,2,0}$	$\tau_{1,2,1}$	$\tau_{1,1,0}$	$\tau_{1,1,1}$
Priority	1	1	2	2
Period	30	40	40	40
Execution Time (in ticks)	9	3	2	2

TABLE II
TASKS IN BOTH SERVERS FOR BOTH MODES.

Task properties and their assignments to the servers is given in Table II. The visualization of the execution of servers and mode changes using different protocols is presented in Figure 7. To ease the understanding of the behavior, we are only presenting servers in these diagrams. In the diagram, the horizontal axis represents the execution time starting from 0. The numbers along the vertical axis are the server's capacity, the diagonal line represents the server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting for its turn to execute (when some other server is executing).



(a) Mode change during normal execution using Suspend-resume protocol (b) Mode change from normal to emergency using Reset/Abort protocol (c) Reset protocol execution using AM

Fig. 7. Testing behaviour of different mode-change protocols.

Figure 7(a) presents the server execution and mode changes using suspend-resume protocol during normal execution in which all tasks and servers are using SRM. The first MCR is raised at time 40 to change system mode from M_0 to M_1, both servers (and their tasks) are suspended and system mode is changed. Later at time 82 another MCR is raised that changes the system mode back to M_0. It is obvious from Figure 7(a) that both servers resume their their execution from where they have previously suspended in mode M_0.

Figures 7(b), and 7(c) present reset (abort) protocol using AM in two different ways. Figures 7(b) describes an example of a mode-change from normal to emergency situation where old modes servers and tasks are aborted and the system starts in a new mode (all servers and tasks are restarted in the new mode). Figures 7(c) describes a reset protocol where the system restarts itself in the same mode (all servers and tasks are restarted in the same mode).

C. *Performance assessments*

Here we present the results of the overhead measurements for transition for the three mechanisms: suspend-resume, abort and complete. It is the time that system spends within the transition state for the mode-switching procedure, i.e. the time spent in changing the current server mode and to update the others servers' and tasks' time values. Complete protocol is notorious for its long mode-change latency due to execution of tasks during transition state. To evaluate these measures, *Complete Protocol* column is added in the tables. These tests have been done for several scenarios, varying the number of tasks and the number of servers. For each value, 100 measures are performed and then average, minimum and maximum of these measured values are presented in Tables III and IV. All measures are in micro-seconds (μs).

Table III provides the overhead measures for different protocols having one server in the system. This test is performed to check the effect of increasing the number of tasks in the server where all tasks are active in all modes. It is clear from measures of Table III that the transition time is increased (roughly linearly) by increasing the number of tasks in a server. It is also obvious that measures for Complete protocol are very high due to the tasks executions during the transition state.

Table IV presents the measurements using more than one servers and tasks in those servers. These results reveal that the execution overhead grows faster by increasing the number of servers than increasing the number of tasks in the system.

VII. Conclusions and Future Work

In this paper, we presented the first implementation of a generic framework for multi-mode adaptation of hierarchical scheduling using the FreeRTOS operating system, supporting multiple modes and multiple mode-change semantics. The framework can be instantiated for a particular mode-change mechanism and can change the mode-change mechanism at runtime. We have developed three basic mechanisms for mode-change semantics for servers as well as for tasks: suspend-resume; complete; and abort. Our setup provides general guidelines and characteristics to develop multiple mode-change protocols. We have illustrate examples to develop different mode-change protocols by composing these mechanisms together and have evaluated these examples by executing the implementation on an EVK1100 board using a 32-bit micro-controller.

The behavioural tests show that the system behaves as expected/specified. The overhead measurements reveal that the execution overhead grows faster by increasing the number of servers in the system as compared with increasing the number of tasks within servers. Since it is the first implementation to support multiple modes in a hierarchical framework, there does not exist any other implementation to compare our results with.

In future we plan to provide schedulability analysis for the protocols. We also plan to make the assumptions more flexible in future, like adding new modes in the system dynamically, providing resource sharing among the tasks of different modes in MMAHSF, etc. Our implementation can be easily incorporated within the ProCom, a component model developed for embedded control systems. The executable components of the ProCom called Runnable Virtual Node [34], executes as periodic server. The user can provide the number of modes, the initial mode change protocol at the modeling level and at the execution level our MMAHSF implementation can be used instead of simple HSF. Hence making ProCom a multi-mode component model.

No. of	Suspend-Resume			Abort			Complete			Complete Protocol		
Tasks	Avg.	Min.	Max	Avg.	Min.	Max	Avg.	Min.	Max	Avg.	Min.	Max
1	173.6	170	181	181.99	181	192	132	128	138	314.06	309	320
2	197.8	192	202	208.49	202	213	159.4	149	160	2535.4	330	9482
4	247.7	245	256	263.6	256	266	229.2	202	256	13634.6	11285	15882

TABLE III
THE OVERHEAD MEASURES FOR DIFFERENT PROTOCOLS HAVING ONE SERVER IN THE SYSTEM.

No. of	No. of	Suspend-Resume			Abort			Complete			Complete Protocol		
Servers	Tasks	Avg.	Min.	Max	Avg.	Min.	Max	Avg.	Min.	Max	Avg.	Min.	Max
2	1	246.87	245	256	271.5	266	277	161.3	160	170	426.86	426	512
4	1	420.2	416	426	455.4	448	458	219.4	213	234	646.75	640	661
3	3	478.99	458	512	508.75	490	533	226.2	234	288	3706.49	661	5856

TABLE IV
THE OVERHEAD MEASURES FOR DIFFERENT PROTOCOLS FOR MULTIPLE SERVERS IN THE SYSTEM.

REFERENCES

[1] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS' 97)*, 1997.

[2] Luca Santinelli, Giorgio C. Buttazzo, and Enrico Bini. Multi-moded resource reservations. In *Proc. 17th IEEE Real-Time Technology and Applications Symposium (RTAS' 11)*, pages 37–46, 2011.

[3] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multimode systems. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS' 10)*, 2010.

[4] V. Nelis, B. Andersson, J. Marinho, and S. M. Petters. Global-EDF scheduling of multimode real-time systems considering mode independent tasks. In *Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS' 11)*, 2011.

[5] N. Fisher and M. Ahmed. Tractable real-time schedulability analysis for mode changes under temporal isolation. In *9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia' 11)*, pages 130–139, 2011.

[6] ATMEL EVK1100 product page. [Online]. Available: http://www.atmel.com/dyn/Products/.

[7] Rafia Inam, Mikael Sjödin, and Reinder J. Bril. Implementing hierarchical scheduling to support multi-mode system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), WiP*, June 2012.

[8] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for hierarchical scheduling in FreeRTOS. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, Tolouse, France, September 2011.

[9] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, 2003.

[10] FreeRTOS web-site. [Online]. Available: http://www.freertos.org/.

[11] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam. Hard real-time support for hierarchical scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, Porto, Portugal, 2011.

[12] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritised preemptive scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, 1986.

[13] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[14] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[15] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS' 06)*, pages 389–398, 2006.

[16] M. Behnam, T. Nolte, M. Sjödin, and I. Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1):93–104, 2010.

[17] Kim Larsen, Paul Pettersson, and Wang Yi. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1:243–264, 1988.

[18] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[19] Q. Guangming. An earlist time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3):181–194, April 2009.

[20] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or EDF scheduling. In *Conference on Design, Automation and Test in Wurope (DATE'09)*, pages 99–104, 2009.

[21] K. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *Proc. 13th IEEE Real-Time Systems Symposium (RTSS' 92)*, 1992.

[22] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *10th Euromicro Conference on Real-Time Systems (ECRTS' 98)*, 1998.

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

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

[25] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis and design language (AADL): An introduction. Technical Report, CMU/SEI-2006-TN-011, 2006.

[26] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A timetriggered language for embedded programming. *Proceedings of the IEEE*, 91(1):166–184, 2003.

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

[28] Hang Yin and Hans Hansson. Timing analysis for mode switch in component-based multi-mode systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS' 12)*, pages 255–264, July 2012.

[29] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS' 02)*, pages 26–35, 2002.

[30] Luca Abeni and Giorgio Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *7th IEEE Real Time Computing Systems and Applications (RTCSA' 99)*, pages 70–77, 1999.

[31] Luca Abeni and Giorgio Buttazzo. Hierarchical QoS management for time sensitive applications. In *7th IEEE Real-Time Technology and Applications Symposium (RTAS' 01)*, pages 63–72, 2001.

[32] N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo. Resource adaptations with servers for hard real-time systems. In *10th International conference on Embedded Software (EMSOFT'10)*, pages 269–278, 2010.

[33] M. Holenderski, R. J. Bril, and J. J. Lukkien. Swift mode changes in memory constrained real-time systems. In *International Conference on Computational Science and Engineering*, 2009.

[34] Rafia Inam. *Towards a Predictable Component-Based Run-Time System.* Number 145 in Mlardalen University Press Licentiate Theses. Licentiate thesis, Mälardalen University Press, Västerås, Sweden, January 2012.

Appendix

A synopsis of the application program interface of MMAHSF implementation is presented below. The names of these API and macros are self-explanatory.

The newly added user API and macro are the following:

1) `void vTaskStartModeScheduler(short defaultMode);`
2) `void vTaskChangeProtocol(short sNewProtocol);`
3) `void vTaskSwitchMode(short sNewMode);`
4) `short sTaskGetCurrentSystemMode(void);`
5) `portBASE_TYPE xTaskIsCompleteInCourse(void);`
6) `short prsReturnTaskArrayIndex(tskTCB *pxTCB);`
7) `short prsReturnServersArrayIndex(subSCB *pxServer);`
8) `unsigned portBASE_TYPE vTaskChangeServerModeBehavior (mode, xBehavior);`
9) `unsigned portBASE_TYPE xTaskChangeTaskModeBehavior (short mode,unsigned portBASE_TYPE xBehavior);`

The newly added private functions and macros are as follows:

1) `void prvMoveTasksToNewMode (short sNewMode, subSCB *pxTempServer);`
2) `void prvMoveCurrentServerAbortProtocol (short sNewMode);`
3) `unsigned portBASE_TYPE prvMoveCurrentServerCompleteProtocol (short sNewMode);`

We adopted the following user APIs to incorporate MMAHSF implementation. The original semantics of these API is kept and used when the user run the original FreeRTOS by setting `configMULTI_MODE` macro to 0.

1) `void vTaskStartScheduler (void);`
2) `portBASE_TYPE xServerCreate(*xServerParameters, *xServerHandle, unsigned portBASE_TYPE *xServerBehaviorMatrix);`
3) `signed portBASE_TYPE xTaskGenericCreate (pxTaskCode, pcName, usStackDepth, *pvParameters, uxPriority, *pxCreatedTask, *puxStackBuffer, xRegions );`
4) `void vTaskDelete (xTaskHandle);`
5) `void vTaskDelay (xTicksToDelay);`
6) `void vTaskDelayUntil (*pxPreviousWakeTime, xTimeIncrement);`
7) `void vTaskPrioritySet (xTaskHandle, *uxNewPriority);`
8) `unsigned portBASE_TYPE uxTaskPriorityGet (xTaskHandle pxTask);`
9) `void vTaskResume (xTaskHandle pxTaskToResume);`
10) `portBASE_TYPE xTaskResumeFromISR (xTaskHandle pxTaskToResume);`
11) `signed portBASE_TYPE xTaskResumeAll (void);`

and adopted private functions and macros:

1) `void vTaskIncrementTick (void);`
2) `void vTaskSwitchContext (void);`
3) `void prvScheduleServers (void);`
4) `signed portBASE_TYPE prxServerInit (*subSCB);`
5) `signed portBASE_TYPE prxRegisterTasktoServer (*tskTCB, *subSCB);`
6) `void prvInitialiseGlobalLists (void);`
7) `void prvInitialiseServerTaskLists (*subSCB);`
8) `void prvInitialiseTCBVariables (*tskTCB, *pcName, *uxPriority, *xRegions, usStackDepth);`
9) `void prvAdjustServerNextReadyTime (*subSCB);`
10) `#define prvAddTaskToReadyQueue (pxTCB);`
11) `#define prvAddServerToReadyQueue (*pxSCB);`
12) `#define prvAddServerToReleaseQueue (*pxSCB);`
13) `#define prvAddTaskToReadyQueue (*pxTCB);`
14) `#define prvChooseNextIdlingServer();`
15) `#define prvCheckDelayedTasks (*pxServer);`