Hard Real-time Support for Hierarchical Scheduling in FreeRTOS*

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Moris Behnam
Mälardalen Real-Time Research Centre
Västerås, Sweden
Email: rafia.inam@mdh.se

***Abstract*—This paper presents extensions to the previous implementation of two-level Hierarchical Scheduling Framework (HSF) for FreeRTOS. The results presented here allow the use of HSF for FreeRTOS in hard-real time applications, with the possibility to include legacy applications and components not explicitly developed for hard real-time or the HSF.**

Specifically, we present the implementations of (i) global and local resource sharing using the Hierarchical Stack Resource Policy and Stack Resource Policy respectively, (ii) kernel support for the periodic task model, and (iii) mapping of original FreeRTOS API to the extended FreeRTOS HSF API. We also present evaluations of overheads and behavior for different alternative implementations of HSRP with overrun from experiments on the AVR 32-bit board EVK1100. In addition, real-time scheduling analysis with models of the overheads of our implementation is presented.

***Index Terms*—real-time systems; hierarchical scheduling framework; resource sharing, fixed-priority scheduling**

I. Introduction

In real-time embedded systems the components and components integration must satisfy both (1) functional correctness and (2) extra-functional correctness, such as satisfying timing properties. Hierarchical Scheduling Framework (HSF) [1] has emerged as a promising technique in satisfying timing properties while integrating complex real-time components on a single node. It supplies an effective mechanism to provide temporal partitioning among components and supports independent development and analysis of real-time systems [2]. In HSF, the CPU is partitioned into a number of subsystems (servers or applications); each real-time component is mapped to a subsystem that contains a local scheduler to schedule the internal tasks of the subsystem. Each subsystem performes its own task scheduling, and the subsystems are scheduled by a global (system-level) scheduler. Two different synchronization mechanisms *overrun* [3] and *skipping* [4] have been proposed and analyzed for inter-subsystem resource sharing, but not much work has been performed for their practical implementations.

We have chosen FreeRTOS [5], a portable open source real-time scheduler to implement hierarchical scheduling framework. The goal is to use the HSF-enabled FreeRTOS to implement the *virtual node* concept in the ProCom component-model [6], [7]. FreeRTOS has been chosen due to its main features, like it's open source nature, small size and scalability, and support of many different hardware architectures allowing it to be easily extended and maintained. Our HSF implementation [8] on FreeRTOS for idling periodic and deferrable servers uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. FPPS is flexible and simple to implement, plus is the de-facto industrial standard for task scheduling. In this paper we extend our implementation of HSF to support hard real-time components. We implement time-triggered periodic tasks within the FreeRTOS operating system. We improve the resource sharing policy of FreeRTOS, and implement support for inter-subsystem resource sharing for our HSF implementation. We also provide legacy support for existing systems or components to be executed within our HSF implementation as a subsystem.

A. Contributions

The main contributions of this paper are:

- We have supported *periodic task model* within the FreeRTOS operating system.
- We have provided *a legacy support* in our HSF implementation and have mapped the old FreeRTOS API to the new API so that the user can very easily use an old system into a server within a two-level HSF.
- We have provided an efficient implementation for *resource sharing* for our HSF implementation. This entails: support for *Stack Resource Policy* for local resource sharing, and *Hierarchical Stack Resource Policy* for global resource sharing with three diferent methods to handle *overrun*.
- We have included the *runtime overhead* for *local and global schedulability analysis* of our implementation.
- We describe the *detailed design* of all the above mentioned improvements in our HSF implementations with the consideration of minimal modifications in underlying FreeRTOS kernel.
- And finally, we have *tested and calculated the performance measures* for our implementations on an AVR-based 32-bit board EVK1100 [9].

B. Resource Sharing in Hierarchical Scheduling Framework

A two-level HSF [10] can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 1. The leaf nodes contain

* This work is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS. Our HSF implementation code is available at http://www.idt.mdh.se/pride/releases/hsf.

its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. The parent node is a global scheduler and is responsible for dispatching the subsystems according to their resource reservations. Using HSF, subsystems can be developed and analyzed in isolation from each other.

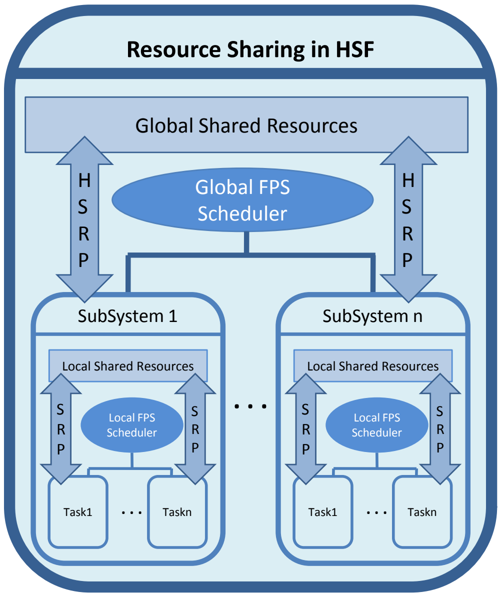

Fig. 1. Two-level Hierarchical Scheduling Framework

In a two-level HSF the resources can be shared among tasks of the same subsystem (or intra-subsystem), normally referred as *local shared resource*. The resources can also be shared among tasks of different subsystems (or inter-subsystem) called *global shared resources* as shown in Figure 1.

Different synchronization protocols are required to share resources at local and global levels, for example, *Stack Resource Policy (SRP)* [11] can be used at local level with FPPS, and to implement SRP-based overrun mechanism at global level, *Hierarchical Stack Resource Policy (HSRP)* [3] can be used.
Organisation: Section II presents the related work on hierarchical scheduler implementations. Section III gives a background on FreeRTOS in III-A, a review of our HSF implementation in FreeRTOS in III-B, and resource sharing techniques in HSF in section III-C. In section IV we provide our system model. We explain the implementation details of periodic task model, legacy support, and resource sharing in section V. In section VI we provide scheduling analysis and in section VII we present the behavior of implementation and some performance measures. In section VIII we conclude the paper. The API for the local and the global resource sharing in HSF is given in Appendix.

II. Related Work

HSF has attained a substantial importance since introduced in 1990 by Deng and Liu [1]. Saewong and Rajkumar [12] implemented and analyzed HSF in CMU's Linux/RK with deferrable and sporadic servers using hierarchical deadline monotonic scheduling. Buttazzo and Gai [13] present an HSF implementation based on Implicit Circular Timer Overflow Handler (ICTOH) using EDF scheduling for an open source RTOS, ERIKA Enterprise kernel. A micro kernel called SPIRIT-μKernel is proposed by Kim *et al.* [10] based on two-level hierarchical scheduling methodology and demonstrate the concept, by porting two different application level RTOS, VxWorks and eCos, on top of the SPIRIT-μKernel. It uses an offline scheduler at global level and the fixed-priority scheduling at local level to schedule the partitions and tasks respectively. A detailed related work on HSF implementation without resource sharing is presented in [8].

A. Local and Global Synchronization Protocols

1) Local synchronization protocols: Priority inheritance protocol (PIP) [14] was developed to solve the priority inversion problem but it does not solve the chained blocking and deadlock problems. Sha *et al.* proposed the priority ceiling protocol (PCP) [14] to solve these problems. A slightly different alternative to PCP is the immediate inheritance protocol (IIP). Baker presented the stack resource policy (SRP) [11] that supports dynamic priority scheduling policies. For fixed-priority scheduling, SRP has the same behavior as IIP. SRP reduces the number of context-switches and the resource holding time as compared to PCP. Like most real-time operating systems, FreeRTOS only support an FPPS scheduler with PIP protocol for resource sharing. We provide support for SRP for local-level resource sharing in HSF.

2) Global synchronization protocols: For global resource sharing some additional protocols have been proposed. Fisher *et al.* proposed Bounded delay Resource Open Environment (BROE) [15] protocol for global resource sharing under EDF scheduling. Hierarchical Stack Resource Policy (HSRP) [3] uses the overrun mechanism to deal with the subsystem budget expiration within the critical section and uses two mechanisms (with pay back and without payback) to deal with the overrun. Subsystem Integration and Resource Allocation Policy (SIRAP) [4] uses the skipping mechanism to avoid the problem of subsystem budget expiration within the critical section. Both HSRP and SIRAP assume FPPS. The original HSRP [3] does not support the independent subsystem development for its analysis. Behnam *et al.* [16] not only extended the analysis for the independent subsystem development, but also proposed a third form of overrun mechanism called extended overrun. In this paper we use HSRP for global resource sharing and implement all the three forms of the overrun protocol.

B. Implementations of Resource Sharing in HSF

Behnam *et al.* [17] present an implementation of a two-level HSF in the commercial operating system VxWorks with the emphasis of not modifying the underlying kernel. The implementation supports both FPS and EDF at both global and local level of scheduling and a one-shot timer is used to trigger schedulers. In [18], they implemented overrun and skipping techniques at the top of their FPS HSF implementation and compared the two techniques.

Holenderski *et al.* [19] implemented a two-level fixed-priority HSF in μC/OS-II, a commercial real-time operating system. This implementation is based on Relative Timed Event Queues (RELTEQ) [20] and virtual timers [21] on the top of RELTEQ to trigger timed events. They incorporated RELTEQ queues and virtual timers within the operating system kernel and provided interfaces for it and HSF implementation uses these interfaces. More recently, they extended the HSF with resource sharing support [22] by implementing SIRAP and HSRP (with and without payback). They measured and compared the system overheads of both primitives.

The work presented in this paper is different from that of [18] in the sense that we implement resource sharing in a two-level HSF with the aim of simplified implementation while adopting the kernel with the consideration of being consistent with the FreeRTOS. The user should be able to choose the original FreeRTOS or HSF implementation to execute, and also able to run legacy code within HSF with doing minimal changes in it. The work of this paper is different from that of [22] in the sense that we only extend the functionality of the operating system by providing support for HSF, and not changing or modifying the internal data structures. It aims at simplified implementation while minimizing the modifications of the underlying operating system. Our implementation is simpler than both [18], [22] since we strictly follow the rules of HSRP [3]. We do not have local ceilings for the global shared resources (as in [18], [22]) which simplifies the implementation. We do not allow local preemptions while holding the global resources which reduces the resource holding times as compared to [18], [22]. Another difference is that both [18], [22] implemented SIRAP and HSRP (with and without payback) while we implement all the three forms of overrun (with payback, without payback, and enhanced overrun). We do not support SIRAP because it is more difficult to use; the application programmer needs to know the WCET of each critical section to use SIRAP. Further neither implementation does provide analysis for their implementations.

III. Background

A. FreeRTOS

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd. It is ported to 23 hardware architectures ranging from 8-bit to 32-bit micro-controllers, and supports many development tools. Its main advantages are portability, scalability and simplicity. The core kernel is simple and small, consisting of three or four (depends on the usage of coroutines) C files with a few assembler functions, 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. Features like ease of use and understandability makes it very popular. More than $77,500$ official downloads in 2009 [23], and the survey result performed by professional engineers in 2010 puts the FreeRTOS at the top for the question "which kernel are you considering using this year" [24] showing its increasing popularity.

FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. In the fixed-priority preemptive scheduling, the tasks with the same priority are scheduled using the round-robin policy. It supports both tasks and subroutines; the tasks with maximum 256 different priorities, any number of tasks and very efficient context switch. FreeRTOS supports both static and dynamic (changed at run-time) priorities of the tasks. It has semaphores and mutexes for resource sharing and synchronization, and queues for message passing among tasks. Its scheduler runs at the rate of one tick per milli-second by default.

FreeRTOS Synchronization Protocol: FreeRTOS supports basic synchronization primitives like *binary, counting* and *recursive semaphore*, and *mutexes*. The mutexes employ *priority inheritance protocol*, that means that when a higher priority task attempts to obtain a mutex that is already blocked by a lower priority task, then the lower priority task temporarily inherits the priority of higher priority task. After returning the mutex, the task's priority is lowered back to its original priority. Priority inheritance mechanism minimizes the *priority inversion* but it cannot cure deadlock.

B. A Review of HSF Implementation in FreeRTOS

A brief overview of our two-level hierarchical scheduling framework implementation [8] in FreeRTOS is given here.

Both global and local schedulers support fixed-priority preemptive scheduling (FPPS). Each subsystem is executed by a server S_s, which is specified by a *timing interface* $S_s(P_s, Q_s)$, where P_s is the period for that server ($P_s > 0$), and Q_s is the capacity allocated periodically to the server ($0 < Q_s \leq P_s$). Each server has a unique priority p_s and a remaining budget during the runtime of subsystem B_s. Since the previous implementation not focus on real-time, we only characterize each task τ_i by its priority ρ_i.

The global scheduler maintains a pointer, running server, that points to the currently running server.

The system maintains two priority-based lists. First is the ready-server list that contains all the servers that are ready (their remaining budgets are greater than zero) and is arranged according to the server's priority, and second is the release-server list that contains all the inactive servers whose budget has depleted (their remaining budget is zero), and will be activated again at their next activation periods and is arranged according to the server's activation times.

Each server within the system also maintains two lists. First is the ready-task list that keeps track of all the ready tasks of that server, only the ready list of the currently running server will be active at any time, and second is the delayed-task list of FreeRTOS that is used to maintain the tasks when they are not ready and waiting for their activation.

The hierarchical scheduler starts by calling `vTaskStartScheduler()` API and the tasks of the highest priority ready server starts execution. At each tick interrupt,

- The system tick is incremented.

- Check for the server activation events. The newly activated server is replenished with its maximum budget and is moved to the ready-server list.
- The global scheduler is called to handle the server events.
- The local scheduler is called to handle the task events.

1) The functionality of the global scheduler: The global scheduler performs the following functionality:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles budget expiration event (i.e. at the budget depletion, the server is moved from the ready-server list to the release-server list).
- Selects the highest priority ready server to run and makes a server context-switch if required. Either `prvChooseNextIdlingServer()` or `prvChooseNextDeferrableServer()` is called to select idling or deferrable server, depending on the value of the `configGLOBAL_SERVER_MODE` macro in the `FreeRTOSConfig.h` file.
- `prvAdjustServerNextReadyTime(pxServer)` is called to set up the next activation time to activate the server periodically.

In idling server, the `prvChooseNextIdlingServer()` function selects the first node (with highest priority) from the ready-server list and makes it the current running server. While in case of deferrable server, the `prvChooseNextDeferrableServer()` function checks in the ready-server list for the next ready server that has any task ready to execute when the currently running server has no ready task even if it's budget is not exhausted. It also handles the situation when the server's remaining budget is greater than 0, but its period ends, in this case the server is replenished with its full capacity.

2) The functionality of the local scheduler: The local scheduler is called from within the tick interrupt using the adopted FreeRTOS kernel function `vTaskSwitchContext()`. The local scheduler is the original FreeRTOS scheduler with the following modifications:

- The round robin scheduling policy among equal priority tasks is changed to FIFO policy to reduce the number of task context-switches.
- Instead of a single ready-task or delayed-task list (as in original FreeRTOS), now the local scheduler accesses a separate ready-task and delayed-task list for each server.

C. Resource sharing in HSF

Stack Resource Policy at global and local levels: We have implemented the HSRP [3] which extends SRP to HSRP. The SRP terms are extended as follows:

- *Priority.* Each task has a priority ρ_i. Similarly, each subsystem has an associated priority p_s.
- *Resource ceiling.* Each globally shared resource R_j is associated with a resource ceiling for global scheduling. This global ceiling is the highest priority of any subsystem whose task is accessing the global resource. Similarly each locally shared resource also has a resource ceiling for local scheduling. This local ceiling is the highest priority of any task (within the subsystem) using the resource.
- *System/subsystem ceilings.* System/subsystem ceilings are dynamic parameters that change during runtime. The system/subsystem ceiling is equal to the currently locked highest global/local resource ceiling in the system/subsystem.

Following the rules of SRP, a task τ_i can preempt the currently executing task within a subsystem only if τ_i has a priority higher than that of running task and, at the same time, the priority of τ_i is greater than the current subsystem ceiling.

Following the rules of HSRP, a task τ_i of the subsystem S_i can preempt the currently executing task of another subsystem S_j only if S_i has a priority higher than that of S_j and, at the same time, the priority of S_i is greater than the current system ceiling. Moreover, whilst a task τ_i of the subsystem S_i is accessing a global resource, no other task of the same subsystem can preempt τ_i.

D. Overrun Mechanisms

This section explains three overrun mechanisms that can be used to handle budget expiry during a critical section in the HSF. Consider a global scheduler that schedules subsystems according to their periodic interfaces . The subsystem budget Q_s is said to expire at the point when one or more internal tasks have executed a total of Q_s time units within the subsystem period P_s. Once the budget is expired, no new task within the same subsystem can initiate its execution until the subsystems budget is replenished at the start of next subsystem period.

To prevent excessive priority inversion due to global resource lock its desirable to prevent subsystem rescheduling during critical sections of global resources. In this paper, we employ the overrun strategy to prevent such rescheduling. Using overrun, when the budget of subsystem expires and it has a task that is still locking a global shared resource, the task continues its execution until it releases the resource. The extra time needed to execute after the budget expiration is denoted as *overrun time* θ. We implement three different overrun mechanisms [16]:

1) The basic overrun mechanism without payback, denoted as BO: here no further actions will be taken after the event of an overrun.
2) The overrun mechanism with payback, denoted as PO: whenever overrun happens, the subsystem S_s pays back in its next execution instant, i.e., the subsystem budget Q_s will be decreased by θ_s i.e. $(Q_s - \theta_s)$ for the subsystems execution instant following the overrun (note that only the instant following the overrun is affected even if $\theta_s > Q_s$).
3) The enhanced overrun mechanism with payback, denoted as EO: It is based on imposing an offset (delaying the budget replenishment of subsystem) equal to the amount of the overrun θ_s to the execution instant

that follows a subsystem overrun, at this instant, the subsystem budget is replenished with $Q_s - \theta_s$.

IV. System Model

In this paper, we consider a two-level hierarchical scheduling framework, in which a global scheduler schedules a system S that consists of a set of independently developed and analyzed subsystems S_s, where each subsystem S_s consists of a local scheduler along with a set of tasks. A system have a set of globally shared resource (lockable by any task in the system), and each subsystem has a set of local shared resource (only lockable by tasks in that subsystem).

A. Subsystem Model

For each subsystem S_s is specified by a subsystem (a.k.a. server) timing interface $S_s = \langle P_s, Q_s, p_s, B_s, X_s \rangle$, where P_s is the period and Q_s is the capacity allocated periodically to the subsystem where $0 < Q_s \leq P_s$ and X_s is the maximum execution-time that any subsystem-internal task may lock a shared global resource. Each server S_s has a unique priority p_s and at each instant during run-time a remaining budget B_s.

It should be noted that X_s is used for schedulability analysis only and our HSRP-implementation does not depend on the availability of this attribute. In the rest of this paper, we use the term subsystem and server interchangeably.

B. Task Model

For hard real-time systems, we are considering a simple periodic task model represented by a set Γ of n number of tasks. Each task τ_i is represented as $\tau_i = \langle T_i, C_i, \rho_i, b_i \rangle$, where T_i denotes the period of task τ_i with worst-case execution time C_i, ρ_i as its priority, and b_i its worst case local blocking. b_i is the longest execution-time inside a critical section with a resource-ceiling equal to or higher than ρ amongst all lower priority task inside the server of τ_i. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. For simplicity, the deadline for each task is equal to T_i.

C. Scheduling Policy

We are using a fixed-priority scheduling FPS at the both global and local level. FPS is the de-facto standard used in industry. For hard-real time analysis we assume unique priorities for each server and unique priorities for each task within a server. However, our implementation support shared priorities, which are then handled in FIFO order (both at global and local scheduling).

D. Design Considerations

Here we present the challenges and goals that our implementation should satisfy:

1) **The use of HSF with resource sharing and the overrun mechanism:** User should be able to make a choice for using the HSF with resource sharing or the simple HSF without using shared resources. Further, user should be able to make a choice for selecting one of the overrun mechanisms, BO, PO, or EO.
2) **Consistency with the FreeRTOS kernel and keeping its API intact:** To embed the legacy code easily within a server in a two-level HSF, and to get minimal changes of the legacy system, it will be good to match the design of implementation with the underlying FreeRTOS operating system. To increase the usability and understandability of HSF implementation for FreeRTOS users, major changes should not be made in the underlying kernel.
3) **Managing local/global system ceilings:** To ensure the correct access of shared resources at both local and global levels, the local and global system ceilings should be updated properly upon the locking and unlocking of those resources.
4) **Enforcement:** Enforcing server execution even at it's budget depletion while accessing a global shared resource; its currently executing task should not be preempted and the server should not be switched out by any other higher priority server (whose `priority` is not greater than the `systemceiling`) until the task releases the resource.
5) **Calculating and deducting overrun time of a server for PO and EO:** In case of payback (PO and EO), the overrun time of the server should be calculated and deducted from the budget at the next server activation.
6) **Protection of shared data structures:** The shared data structures that are used to lock and unlock both local and global shared resources should be accessed in a mutual exclusive manner with respect to the scheduler.

V. Implementation

A. Support for Time-triggered Periodic Tasks

Since we are following the periodic resource model [25], we need the periodic task behavior implemented within the operating system. Like many other real-time operating systems, FreeRTOS does not directly support the periodic task activation. We incorporated the periodic task activation as given in Figure 2. To do minimal changes in the underlying operating system and save memory, we add only one additional variable `readyTime` to the task TCB, that describes the time when the task will become ready. A user API `vTaskWaitforNextPeriod(period)` is implemented to activate the task periodically. The FreeRTOS delayed-task list used to maintain the periodic tasks when they are not ready and waiting for their next activation period to start. Since FreeRTOS uses ticks, period of the task is given in number of ticks.

```
// task function
while (TRUE) do {
    taskbody();
    vTaskWaitforNextPeriod(period);
end while
```

Fig. 2. Pseudo-code for periodic task model implementation

B. Support for Legacy System

To implement legacy applications support in HSF implementation for the FreeRTOS users, we need to map the original FreeRTOS API to the new API, so that the user can run its old code in a subsystem within the HSF. A macro `configHIERARCHICAL_LEGACY` must be set in the config file to utilize legacy support. The user should rename the old `main()` function, and remove the `vTaskStartScheduler()` API from legacy code.

The legacy code is created in a separate server, and in addition to the server parameters like period, budget, priority, user also provides a function pointer of the legacy code (the old main function that has been renamed). `xLegacyServerCreate(period, budget, priority, *serverHandle, *functionPointer)` API is provided for this purpose. The function first creates a server and then creates a task called `vLegacyTask(*functionPointer)` that runs only once and performs the initialization of the legacy code (executes the old main function which create the initial set of tasks for the legacy application), and destroys itself. When the legacy server is replenished first time, all the tasks of the legacy code are created dynamically within the currently running legacy server and start executing.

We have adopted the original FreeRTOS `xTaskGenericCreate` function to provide legacy support. If `configHIERARCHICAL_SCHEDULING` and `configHIERARCHICAL_LEGACY` macros are set then `xServerTaskGenericCreate` function is called that creates the task in the currently executing server instead of executing the original code of `xTaskGenericCreate` function.

This implementation is very simple and easy to use, user only needs to rename old `main()`, remove `vTaskStartScheduler()` from legacy code, and use a single API to create the legacy server. It should be noted that the HSF guarantees separation between servers; thus a legacy non/soft real-time server (which e.g. is not analyzed for schedulability or not use predictable resource locking) can co-exists with hard real-time servers.

C. Support for Resource sharing in HSF

Here we describe the implementation details of the resource sharing in two-level hierarchical scheduling framework. We implement the local and global resource sharing as defined by Davis and Burns [3]. For local resource sharing SRP is used and for global resource sharing HSRP is used. Further all the three forms of overrun as given by Behnam *et al.* [16] are implemented. The resource sharing is activated by setting the macro `configGLOBAL_SRP` in the configuration file.

1) Support for SRP: For local resource sharing we implement SRP to avoid problems like priority inversions and deadlocks.

The data structures for the local SRP: Each local resource is represented by the structure `localResource` that stores the resource ceiling and the task that currently holds the resource as shown in Figure 3. The locked resources are stacked onto the `localSRPList`; the FreeRTOS list structure is used to implement the SRP stack. The list is ordered according to the resource ceiling, and the first element of list has the highest resource ceiling, and represents the local `system ceiling`.

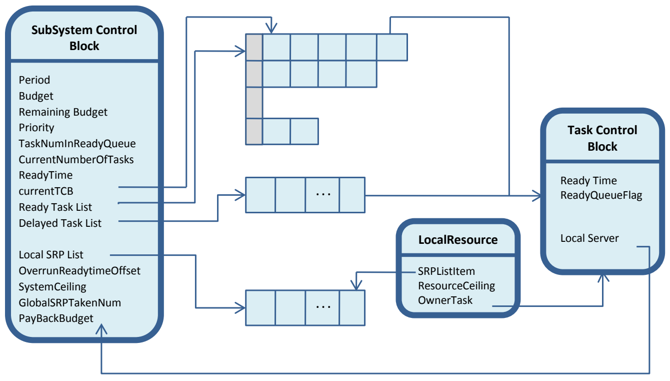

Fig. 3. Data structures to implement SRP

The extended functionality of the local scheduler with SRP: The only functionality extended is the searching for the next ready task to execute. Now the scheduler selects a task to execute if the task has the highest priority among all the ready tasks and its priority is greater than the current system ceiling, otherwise the task that has locked the highest (top) resource in the `localSRPList` is selected to execute. The API list for the local SRP is provided in the Appendix.

2) Support for HSRP: HSRP is implemented to provide global resource sharing among servers. The resource sharing among servers at the global level can be considered the same as sharing local resources among tasks at the local level. The details are as follows:

The data structures for the global HSRP: Each global resource is represented by the structure `globalResource` that stores the global-resource ceiling and the server that currently holds the resource as shown in Figure 4. The locked resources are stacked onto the `globalHSRPList`; the FreeRTOS list structure is used to implement the HSRP stack. The list is ordered according to the resource ceiling, the first element of the list has the highest resource ceiling and represents the `GlobalSystemCeiling`.

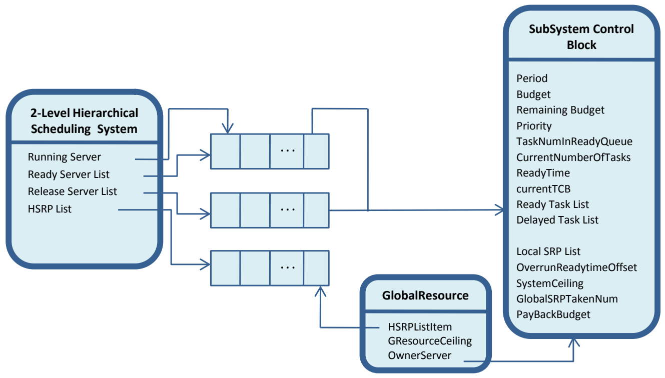

Fig. 4. Data structures to implement HSRP

The extended functionality of the global scheduler with HSRP: To incorporate HSRP into the global scheduler, `prvChooseNextIdlingServer()` and `prvChooseNextDeferrableServer()` macros are appended with the following functionality: The global scheduler selects a server if the server has the highest priority among all the ready servers and the server's priority is greater than the current `GlobalSystemCeiling`, otherwise the server that has locked the highest(top) resource in the `HSRPList` is selected to execute. The API list for the global HSRP is provided in Appendix.

3) Support for Overrun Protocol: We have implemented three types of overrun mechanisms; without payback (BO), with payback (PO), and enhanced overrun (EO). Implementation of BO is very simple, the server simply executes and overruns its budget, and no further action is required. For PO and EO we need to measure the overrun amount of time to pay back at the server's next activation.

The data for the PO and EO Overrun mechanisms: Two variables `PayBackBudget` and `OverrunReadytimeOffset` are added to the subsystem structure `subSCB` to keep a record of the overrun amount to be deducted from the next budget of the server as shown in Figure 4. The overrun time is measured and stored in `PayBackBudget`. `OverrunReadytimeOffset` is used in EO mechanism to impose an offset in the next activation of server.

The extended functionality of the global scheduler with Overrun: A new API `prvOverrunAdjustServerNextReadyTime(*pxServer)` is used to embed overrun functionality (PO and EO) into the global scheduler. For both PO and EO, the amount of overrun, i.e. `PayBackBudget` is deducted from the server `RemainingBudget` at the next activation period of the server, i.e. $B_s = Q_s - \theta_s$. For EO, in addition to this, an offset (O_s) is calculated that is equal to the amount of overrun, i.e. $O_s = \theta_s$. The server's next activation time (the budget replenishment of subsystem) is delayed by this offset. `OverrunReadytimeOffset` variable is used to store the offset for next activation of the server.

4) Safety Measure: We have modified `vTaskDelete` function in order to prevent the system from crashing when users delete a task which still holds a local SRP or a global HSRP resource. Now it also executes two private functions `prvRemoveLocalResourceFromList (*pxTaskToDelete)`, and `prvRemoveGlobalResourceFromList (*pxTaskToDelete)`, before the task is deleted.

D. Addressing Design Considerations

Here we address how we achieve the design requirements that are presented in Section IV-D.

1) **The use of HSF with resource sharing and the overrun mechanism:** The resource sharing is activated by setting the macro `configGLOBAL_SRP` in the configuration file. The type of overrun can be selected by setting the macro `configOVERRUN_PROTOCOL_MODE` to one of the three values: `OVERRUN_WITHOUT_PAYBACK`, `OVERRUN_PAYBACK`, or `OVERRUN_PAYBACK_ENHANCED`.
2) **Consistency with the FreeRTOS kernel and keeping its API intact:** We have kept consistence with the FreeRTOS from the naming conventions to API, data structures and the coding style used in our implementations; for example all the lists used in our implementation are maintained in a similar way as of FreeRTOS.
3) **Managing local/global system ceilings:** The correct access of the shared resources at both local and global levels is implemented within the functionality of the API used to lock and unlock those resources.
When a task locks a local/global resource whose ceiling is higher than the subsystem/system ceiling, the resource mutex is inserted as the first element onto the `localSRPList`/`HSRPList`, the `systemceiling` /`GlobalSystemCeiling` is updated, and this task/server becomes the owner of this local/global resource respectively. Each time a global resource is locked, the `GlobalResourceTakenNum` is incremented.
Similarly upon unlocking a local/global resource, that resource is simply removed from the top of the `localSRPList`/`HSRPList`, the `systemceiling` /`GlobalSystemCeiling` is updated, and the owner of this resource is set to `NULL`. For global resource, the `GlobalResourceTakenNum` is decremented.
4) **Enforcement:** `GlobalResourceTakenNum` is used as an overrun flag, and when its value is greater than zero (means a task of the currently executing server has locked a global resource), no other higher priority server (whose `priority` is not greater than the `systemceiling`) can preempt this server even if its budget depletes.
5) **Overrun time of a server for PO and EO:** `prvOverrunAdjustServerNextReadyTime` API is used to embed the overrun functionality into the global scheduler as explained in section V-C3.
6) **Protection of shared data structures:** All the functionality of the APIs (for locking and unlocking both local and global shared resources) is executed within the FreeRTOS macros `portENTER_CRITICAL()` and `portEXIT_CRITICAL()` to protect the shared data structures.

VI. Schedulability analysis

This section presents the schedulability analysis of the HSF, starting with local schedulability analysis (i.e. checking the schedulability of each task within a server, given the servers timing interface), followed by global schedulability analysis (i.e., checking that each server will receive its capacity within its period given the set of all servers in a system).

A. The Local Schedulability Analysis

The local schedulability analysis can be evaluated as follows [25]:

$$\forall \tau_i \; \exists t : 0 < t \leq D_i, \;\; \mathtt{rbf}(i,t) \leq \mathtt{sbf}(t), \tag{1}$$

where `sbf` is the supply bound function, based on the periodic resource model presented in [25], that computes the minimum possible CPU supply to S_s for every time interval length t, and $\texttt{rbf}(i,t)$ denotes the request bound function of a task τ_i which computes the maximum cumulative execution requests that could be generated from the time that τ_i is released up to time t and is computed as follows:

$$\texttt{rbf}(i,t) = C_i + b_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{2}$$

where $\texttt{HP}(i)$ is the set of tasks with priorities higher than that of τ_i and b_i is the maximum local blocking.

The evaluation of sbf depends on the type of the overrun mechanism;

a) Overrun without payback:

$$\texttt{sbf}(t) = \begin{cases} t-(k+1)(P_s-Q_s) & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{3}$$

where $k = \max\Big(\big\lceil\big(t-(P_s-Q_s)\big)/P_s\big\rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$.

b) Overrun with payback [16]:

$$\texttt{sbf}(t) = \max\Big(\min\Big(f_1(t), f_2(t)\Big), 0\Big), \tag{4}$$

where $f_1(t)$ is

$$f_1(t) = \begin{cases} t-(k+1)(P_s-Q_s)-X_s & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{5}$$

where $k = \max\Big(\big\lceil\big(t-(P_s-Q_s)-X_s\big)/P_s\big\rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s + X_s, (k+1)P_s - Q_s + X_s]$, and $f_2(t)$ is

$$f_2(t) = \begin{cases} t-(2)(P_s-Q_s) & \text{if } t \in V^{(k)} \\ t-(k+1)(P_s-Q_s)-X_s & \text{if } t \in Z^{(k)} \\ (k-1)Q_s - X_s & \text{otherwise,} \end{cases} \tag{6}$$

where $k = \max\Big(\big\lceil\big(t-(P_s-Q_s)\big)/P_s\big\rceil, 1\Big)$, $V^{(k)}$ denotes an interval $[2P_s - 2Q_s, 2P_s - Q_s - X_s]$, and $Z^{(k)}$ denotes an interval $[(k+2)P_s - 2Q_s, (k+2)P_s - Q_s]$.

c) Enhanced overrun:

$$\texttt{sbf}(t) = \max\Big(f_2(t), 0\Big). \tag{7}$$

B. The Global Schedulability Analysis

A global schedulability condition is

$$\forall S_s \; \exists t : 0 < t \le P_s, \; \texttt{RBF}_s(t) \le \texttt{t}. \tag{8}$$

where $\texttt{RBF}_s(t)$ is the request bound function and it is evaluated depending on the type of server (deferrable or idling) and type of the overrun mechanism (see [16] for more details). First, we will assume the idling server and later we will generalize our analysis to include deferrable server.

d) Overrun without payback:

$$\texttt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \texttt{HPS}(s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot (Q_k + X_k). \tag{9}$$

where $\texttt{HPS}(s)$ is the set of subsystems with priority higher than that of S_s. Let Bl_s denote the maximum blocking imposed to a subsystem S_s, when it is blocked by lower-priority subsystems.

$$Bl_s = \max\{X_j | \; S_j \in \texttt{LPS}(S_s)\}, \tag{10}$$

where $\texttt{LPS}(S_s)$ is the set of subsystems with priority lower than that of S_s.

e) Overrun with payback:

$$\texttt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \texttt{HPS}(s)} \left(\left\lceil \frac{t}{P_k} \right\rceil (Q_k) + X_k \right). \tag{11}$$

f) Enhanced overrun:

$$\texttt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \texttt{HPS}(s)} \left(\left\lceil \frac{t + J_k}{P_k} \right\rceil (Q_k) + X_k \right). \tag{12}$$

Where $J_s = X_s$ and the schedulability analysis for this type is

$$\forall S_s, 0 < \exists t \le P_s - X_s, \; \texttt{RBF}_s(t) \le \texttt{t}, \tag{13}$$

For deferrable server, a higher priority server may execute at the end of its period and then at the beginning of the next period. To model such behavior a jitter (equal to $P_k - (O_k + X_k)$) is added to the ceiling in equations 9, 11 and 12.

C. Implementation Overhead

In this section we will explain how to include the implementation overheads in the global schedulability analysis.

Looking at the implementation we can distinguish two types of runtime overhead associated with the system tick: (1) a repeated overhead every system tick independently if it will release a new server or not, and (2) an overhead which occurs whenever a server is activated and it includes the overhead of scheduling, maybe context switch, budget depletion after consuming the budget then another context switch and scheduling and finally it includes the overrun overhead.

(1) Is called fixed overhead (fo) and it is the result of updating the system tick and perform some checking and its value is always fixed. This overhead can be added to equation 8. This equation assumes that the processor can provide all CPU time to the servers (t in the right side of the equation) now we assume that every system tick (st), a part will be consumed by the operating system (fo) and then instead of using t in the right side of equation 8, we can use $(1 - fo/st) \times t$ to include the fixed overhead. (st defaults to $1ms$ for our implementation.)

(2) Is called server overhead (so) and repeats periodically for every server, i.e. with a period P_i. Since the server

overhead is executed by the kernel its not enough to model it as extra execution demand from the server. Instead the overhead should be modeled as a separate server S_o (one server S_o corresponding to each real server S_i) executing at a priority higher that of any real server with parameters $P_o = P_i$, $Q_o = so$, and $X_o = 0$.

The overhead-parameters are dependent on the number of servers, tasks and priority levels, etc. and should be quantified with static WCET-analysis which is beyond the scope of this paper; however some small test cases reported in [8] the measured worst-case for idling servers are $fo = 32\mu s$ and $so = 74\mu s$, and for deferrable servers they are $fo = 32\mu s$ and $so = 85\mu s$ for three servers with total seven tasks.

VII. Experimental Evaluation

In this section, we report the evaluation of behavior and performance of the resource sharing in HSF implementation. All measurements are performed on the target platform EVK1100 [9]. The AVR32UC3A0512 micro-controller runs at the frequency of 12MHz and its tick interrupt handler at 1ms.

A. Behavior Testing

In this section we perform an experiment to test the behavior of overrun in case of global resource sharing in HSF implementation. The experiment is performed to check the overrun behavior in idling periodic server by means of a trace of the execution. Two servers S1, and S2 are used in the system, plus idle server is created. The servers used to test the system are given in Table I.

Server	S1	S2
Priority	2	1
Period	20	40
Budget	10	15

TABLE I
Servers used to test system behavior.

Note that higher number means higher priority. Task properties and their assignments to the servers is given in Table II. $T2$ and $T3$ share a global resource. The execution time of $T2$ is $(3+3)$ that means a normal execution for initial 3 time units and the critical section execution for the next 3 time units, similarly $T3$ $(10+8)$ executes for 10 time units before critical section and executes for 8 time units within critical section. The visualization of the executions of budget overrun without payback (BO) and with payback (PO) for idling periodic server are presented in Figure 5 and Figure 6 respectively.

Tasks	T1	T2	T3
Servers	$S1$	$S1$	$S2$
Priority	2	1	1
Period	15	20	60
Execution Time	3	$(3+3)$	$(10+9)$

TABLE II
Tasks in both servers.

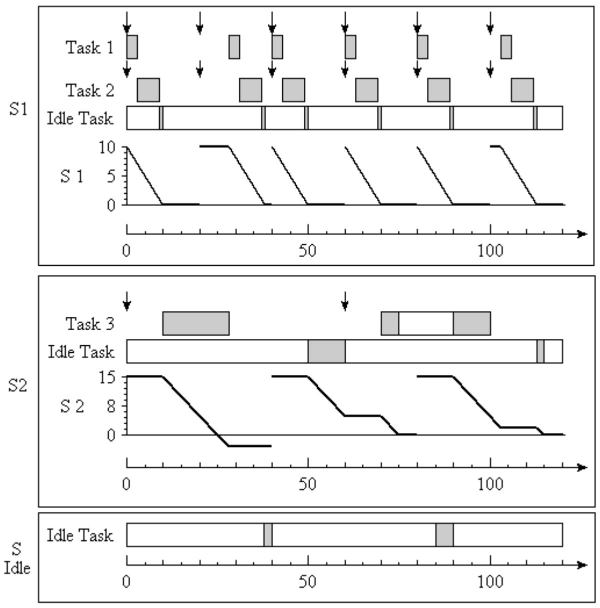

Fig. 5. Trace of budget overrun without payback (BO) for Idling server

In the visualization, the arrow represents task arrival, a gray rectangle means task execution. In Figure 5 at time 20, the high priority server $S1$ is replenished, but its priority is not higher than the global system ceiling, therefore, it cannot preempt server $S2$ which is in the critical section. $S2$ depletes its budget at time 25, but continues to executes in its critical section until it unlocks the global resource at time 29. The execution of S1 is delayed by 9 time units.

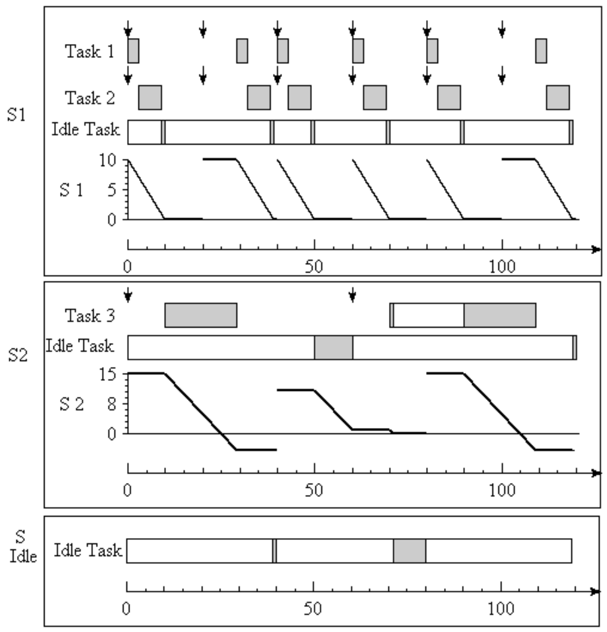

Fig. 6. Trace of budget overrun with payback (PO) for Idling server

In case of overrun with payback, the overrun time is deducted from the budget at the next server activation, as shown in Figure 6. At time 40 the server $S2$ is replenished with a reduced budget, while in case of overrun without payback the server is always replenished with its full budget as obvious from Figure 5.

B. Performance Measures

Here we report the performance measures of lock and unlock functions for both global and local shared resources.

The execution time of functions to lock and unlock global and local resources is presented in Table III. For each measure, a total of 1000 values are computed. The minimum, maximum, average and standard deviation on these values are calculated and presented for both types of resource sharing.

Function	Min.	Max.	Average	St. Dev.
`vGlobalResourceLock`	21	21	21	0
`vGlobalResourceUnlock`	32	32	32	0
`vLocalResourceLock`	21	32	26.48	5.51
`vLocalResourceUnlock`	21	21	21	0

TABLE III
THE EXECUTION TIME (IN MICRO-SECONDS (μS)) OF GLOBAL AND LOCAL LOCK AND UNLOCK FUNCTION.

VIII. CONCLUSIONS

In this paper, we have provided a hard real-time support for a two-level HSF implementation in an open source real-time operating system FreeRTOS. We have implemented the periodic task model within the FreeRTOS kernel. We have provided a very simple and easy implementation to execute a legacy system in the HSF with the use of a single API. We have added the SRP to the FreeRTOS for efficient resource sharing by avoiding deadlocks. Further we implemented HSRP and overrun mechanisms (BO, PO, EO) to share global resources in a two-level HFS implementation. Under assumption of nested locking, the overrun is bounded and is equal to the longest resource-holding time. Hence, the temporal isolation of HSF is subject to the bounded resource-holding time.

We have focused on doing minimal modifications in the kernel to keep the implementation simple and keeping the original FreeRTOS API intact. We have presented the design and implementation details and have tested our implementations on the EVK1100 board. We have included the overheads for local-level and global-level resource sharing into the schedulability analysis. In future we plan to integrate the virtual node concept of ProCom model on-top of the presented HSF [6], [7].

REFERENCES

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

[2] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. *Component-Based Software engineering*, LNCS-3054(2004):209–216, May 2005.

[3] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*.

[4] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Sjödin. SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proc. EMSOFT*, pages 279–288, October 2007.

[5] FreeRTOS web-site. http://www.freertos.org/.

[6] J. Carlsson, J. Feljan, and M. Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *36th (SEAA)*, 2010.

[7] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *WiP Session of (ECRTS10)*, pages 17–20, 2010.

[8] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Support for hierarchical scheduling in FreeRTOS. In *To appear in the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA11)*, September 2011.

[9] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/.

[10] Daeyoung Kim, Yann-Hang Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proceedings (RTCSA00)*, 2000.

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

[12] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *IEEE (RTSS01)*, 2001.

[13] G. Buttazzo and P. Gai. Efficient edf implementation for small embedded systems. In *International Workshop on (OSPERT06)*, 2006.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *Journal of IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[15] N. Fisher, M. Bertogna, and S. Baruah. The design of an edf-scheduled resource-sharing open environment. In *IEEE (RTSS07)*.

[16] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE TII*, 6(1), February 2010.

[17] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop (OSPERT'08)*.

[18] Mikael Åsberg, Moris Behnam, Thomas Nolte, and Reinder J. Bril. Implementation of overrun and skipping in vxworks. In *Proceedings of the 6th International Workshop (OSPERT10)*, 2010.

[19] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Extending an Open-source Real-time Operating System with Hierarchical Scheduling. Technical Report, Eindhoven University, 2010.

[20] M. Holenderski, W. Cools, Reinder J. Bril, and J. J. Lukkien. Multiplexing Real-time Timed Events. In *Work in Progress session of (ETFA09)*.

[21] M.M.H.P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien. Virtual Timers in Hierarchical Real-time Systems. In *Work in Progress Session of (RTSS09)*, December 2009.

[22] M.M.H.P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien. Extending an HSF-enabled Open-Source Rel-Time Operating System with Resource sharing. In *(OSPERT10)*, 2010.

[23] Microchip web-site.

[24] EE TIMES web-site. http://www.eetimes.com/design/embedded/4008920/The-results-for-2010-are-in-.

[25] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *IEEE (RTSS03)*, pages 2–13, 2003.

APPENDIX

A synopsis of the application program interface to implement resource sharing in HSF implementation is presented below. The names of these API are self-explanatory.

1) `xLocalResourcehandle xLocalResourceCreate(uxCeiling)`
2) `void vLocalResourceDestroy(xLocalResourcehandle)`
3) `void vLocalResourceLock(xLocalResourcehandle)`
4) `void vLocalResourceUnLock(xLocalResourcehandle)`
5) `xGlobalResourcehandle xGlobalResourceCreate (uxCeiling)`
6) `void vGlobalResourceDestroy(xGlobalResourcehandle)`
7) `void vGlobalResourceLock(xGlobalResourcehandle)`
8) `void vGlobalResourceUnLock(xGlobalResourcehandle)`