Extending μCOS-II with FPDS and Reservations

Wim Cools

Master Thesis
2IM90
July 21, 2010

System Architecture and Networking
Department of Mathematics and Computer Science
Eindhoven University of Technology

Supervisors

dr. ir. R.J. Bril
M. Holenderski MSc.

Abstract

To guarantee processor resources under fluctuating network bandwidth, the usage of reservations and fixed-priority scheduling with deferred preemption (FPDS) has been investigated [30] for an application in the surveillance domain. Our goal is to design and implementation of reservations and FPDS in the μC/OS-II real-time operating system. The design should be efficient, extensible and modular, while minimizing modifications of the underlying operating system.

Resource reservations are used to virtualize computing resources and guarantee access of a requested share of this resource. We use processor reservations to guarantee processor time to real-time subsystems that share a single processor. Support for processor reservations depends on four ingredients: *admission*, *scheduling*, *accounting,* and *enforcement.* While admission is done offline, we use a two-level hierarchical scheduling framework (HSF) for scheduling, and servers for accounting and enforcement. We provide support for deferrable, periodic and polling servers. The HSF consists of a global scheduler to schedule the servers, and a local scheduler to schedule tasks within a server.

We use a Fixed Priority Preemptive Scheduling (FPPS) mechanism both on a global and local level. To reduce the overhead of arbitrary preemption points, and to guard critical sections in which shared resources can be accessed, we extend the scheduler to support FPDS. To support the periodic task model, periodic task primitives are also implemented.

As an FPDS subjob cannot be preempted, when FPDS is used in combination with reservations, some mechanism is required to prevent the depletion of the reservation's share of CPU time (server budget) while the subjob is executing. We base our solution, H-FPDS, on the skipping and overrun mechanisms, used by synchronization protocols.

We identified the need for a component to deal with periodic arrival, depletion, replenishment and other timed events. To this end we propose the design of RELTEQ, a versatile timed event management component targeted at embedded operating systems. Our RELTEQ extension provides mechanisms for (1) multiplexing timers on a single hardware timer, (2) avoiding interference of events from inactive subsystems, and (3) virtual timers.

In this thesis we have presented an efficient, extensible and modular design for extending a real-time operating system with periodic tasks, FPDS, and reservations. It relies on RELTEQ, a component to manage timed events in embedded operating systems. We have successfully implemented our design in the μC/OS-II real-time operating system, while minimizing modifications to the kernel. We evaluated our design based on overhead, behavior and modularity. Preliminary results of this work were published in [33, 32], and our work is used as a basis for further extensions [34, 44].

Contents

1 Introduction

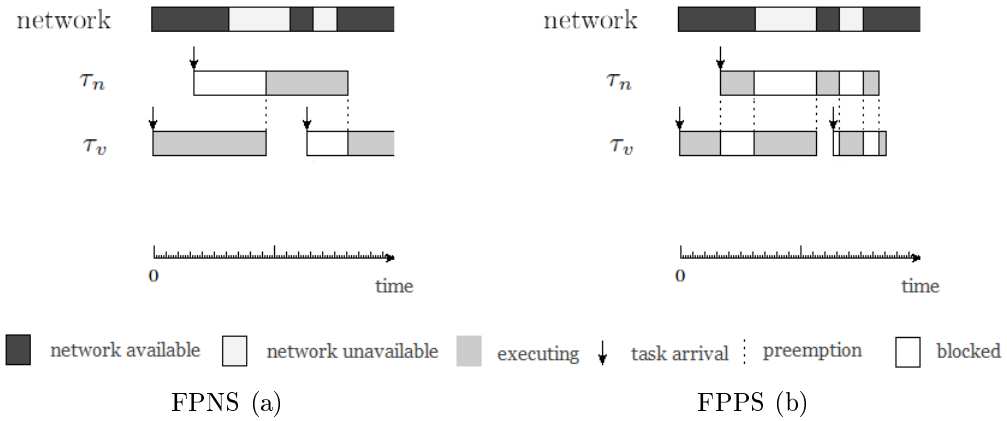
While operating in a resource constrained environment, the functions performed by embedded real-time systems become increasingly complex. Despite limited and fluctuating availability of resources shared by tasks running on such embedded platforms, their correct timely behavior must be guaranteed. Resource reservation and Fixed Priority Scheduling with Deferred Preemption (FPDS) can be used as a low-overhead mechanism for providing resource guarantees. In this thesis we focus on the design and implementation of these mechanisms in μC/OS-II [26], a real-time operating system for embedded systems. In this chapter we will provide a short introduction to the research domain, and motivate the use of resource reservations. We then list our goals for this thesis, and the approach we have taken to come to the contributions.

1.1 Context & Background

As part of the ITEA/CANTATA project [3], our group is investigating the real-time aspects of a multimedia processing system from the surveillance domain, designed by an industrial partner. The system contains a camera platform which monitors for example a bank office, equipped with an embedded processing platform. The platform runs two main periodic tasks: a video task τ_v and a network task τ_n. The video task is responsible for loading the raw video frames from main memory, analyze the video content (e.g. detect suspicious movements), encode the frames and place the result back in main memory. The network task loads the encoded frames from the main memory, converts them into network packets, and sends them over the network interface (e.g. to a police station). The memory transfers are performed via DMA (*Direct Memory Access*) over a shared bus. Each task is assigned a period (defined by the video stream rate), and a priority, with the priority of τ_n higher than τ_v.

As described in [30], the platform currently cannot make optimal use of the processor. For data intensive tasks, typical in the domain of streaming multimedia, context switching (switching execution from one task to another) can be very expensive in terms of time overhead. For example, when a data transfer to a resource (e.g. network or memory) is interrupted, it might have to be retransmitted or access to the resource reinitialized (e.g. bus access for DMA). Due to the high cost of context switches, the tasks are currently scheduled according to FPNS (*Fixed Priority Non-Preemptive Scheduling)*, which means a task that is currently running cannot be preempted when a higher priority task arrives. Using FPNS, however, the platform cannot make optimal use of the processor: as the network availability is fluctuating, τ_n might be scheduled in when the network is congested, or the video task might be executing when the network is available (see Figure 1.a). If the cost of context switches was low, we could use FPPS (*Fixed Priority Preemptive Scheduling*), where a task can be preempted at an abitrary point, and τ_v will thus be preempted whenever τ_n is ready (see Figure 1.b). Instead, [30] suggests FPDS (*Fixed Priority Scheduling with Deferred Preemption*) as a suitable scheduling mechanism. FPDS takes the middle ground between FPNS and FPPS, by allowing preemptions before a job is completed, but only at certain pre-defined *preemption points*. These points are placed at places which are convenient for a task, i.e. where the overhead of context switching sufficiently small. This will allow the network task to be

activated more frequently, but only at times when the cost of a context switch is low (e.g. when a DMA data transfer has been completed). By placing the preemption points around the access to a shared resource, we can also use it to guard critical sections, and therefore also prevent the need for resource access protocols, further reducing the overhead.



FPNS (a) FPPS (b)

Figure 1: Network availability and execution trace of τ_n and τ_v.

As the video task can now be preempted by the network task, we use reservations [8] to guarantee processor resources to the video task. Using resource reservations, resources (like the processor) are virtualized and each set of tasks is assigned its own share of the resource. By using processor reservations we provide *temporal isolation* amongst sets of tasks: every period, each set of task is guaranteed a certain amount of processor time.

1.2 Problem description & Goals

Resource reservations and FPDS were identified as mechanisms to guarantee processor usage despite fluctuating availability of resources [30]. We want to add this functionality to μC/OS-II, the operating system used by our industrial and academic partners. To make sure our design is maintainable, efficient and reusable for future extensions, exploring design alternatives and making architectural trade-offs plays an important part. We set the following requirements for our design:

- Supports the creation and management of processor reservations, and FPDS scheduling.
- Modular: parts of our extension can be switched on and off during compile time, or disabled alltogether.
- Extensible and reusable: our design can be used for future extensions and by other applications.
- Efficient: low runtime overhead and small memory footprint.
- μC/OS-II compatibility: use existing data structures when possible, with minimal changes to the μC/OS-II kernel.

1.3 Approach

Our first step was to investigate the existing literature to identify the ingredients needed to implement FPDS and reservations. By identifying these (periodic tasks, servers, and hierarchical scheduling) and comparing their existing implementations in other related works, we found timed event management to be a building block which could be shared by all these components. We then designed the timer management component, and used it to replace and extend the existing μC/OS-II timers. Measurements were performed by introducing profiling functions to μC/OS-II, and running our design on the OpenRISC 1000 simulator [29]. Based on the timed event manager, we designed and evaluated extensions for periodic tasks, servers, hierarchical scheduling, and FPDS with reservations.

1.4 Contributions

The contributions of this thesis are as follows:

- The design of an efficient timed event management system (RELTEQ) for embedded systems, with support for virtual timers and mechanisms to avoid interference of events from inactive subsystems.
- A modular design and implementation for periodic tasks and reservations in μC/OS-II based on RELTEQ. Reservations are based on the deferrable, periodic, and polling servers.
- Design and implementation of a *Hierarchical Scheduling Framework* (HSF), where a global scheduler schedules the reservations globally, which in turn use a local scheduler to schedule the tasks sharing the reservation locally.
- Support for FPDS, and the combination of reservations with FPDS scheduling (H-FPDS).
- An evaluation of the implementation of all our extensions, in terms of behavior, efficiency, memory footprint, and modularity.

Preliminary results of this work were published in [33] and [32]. Our extensions have also been used as a basis of further work on supporting synchronization protocols in μC/OS-II in [34] and [44].

1.5 Overview

In this chapter we briefly stated the context and motivation of our project, and the problem we try to solve.

In Chapter 2 we explore the context in detail, and explain the terms, concepts and system model relevant to the domain of this thesis, which is a prerequisite for the discussion in the remaining chapters. We also show related work and existing implementations, and compare it to the approach we take.

Chapter 3 explains the real-time operating system μC/OS-II, its architecture and implementation details relevant for our design. We also describe the platform on which we ran our simulations, and how we perform measurements inside the μC/OS-II kernel.

In Chapters 4 to 7, we introduce our extensions, one by one. Each chapter starts with with a motivation and describes the functionality we wish to offer. We then describe the architecture, design and finally the implementation based on previously introduced extensions. At the end of each chapter we evaluate the extension in terms of efficiency (time and space overhead) and modularity. The extensions are: FPDS (Ch. 4), the timed event manager RELTEQ (Ch. 5), Periodic tasks (Ch. 6), and Reservations (Ch. 7).

Finally we conclude in Chapter 8, and present possibilities for future work.

1.6 Glossary

Notation	Description
API	*Application Programming Interface*
DMA	*Direct Memory Access*
EDF	*Earliest Deadline First*
FPDS	*Fixed Priority Scheduling with Deferred Preemption*
FPNS	*Fixed Priority Non-Preemptive Scheduling*
FPPS	*Fixed Priority Preemptive Scheduling*
FPS	*Fixed-Priority Scheduling*
HPET	*High Precision Event Timer*
HSF	*Hierarchical Scheduling Framework*
HSRP	*Hierarchical Stack Resource Policy*
ISR	*Interrupt Service Routine*
LOC	*Lines Of Code*
POSIX	*Portable Operating System Interface*
QoS	*Quality of Service*
RAM	*Random-Access Memory*
RELTEQ	*RELative Timed Event Queues*
ROM	*Read-Only Memory*
SCB	*Server Control Block*
SDK	*Software Development Kit*
SIRAP	*Subsystem Integration and Resource Allocation Policy*
TCB	*Task Control Block*
WCET	*Worst-Case Execution Time*

2 Theory & Related Work

In this chapter, we provide an overview of existing theory and research that is related to our topic. We discuss related literature on the topics of real-time operating systems, scheduling, and reservations, and look at the existing implementations, all of which form the basis for this thesis.

2.1 Real-Time Systems

The main distinction between a real-time system and other systems, is that correct execution not only depends on functional requirements, but is also subject to temporal constraints. Even if the result of a task is correct from a computational standpoint, if it fails to finish within a given deadline, the result can still be useless or wrong. On an average desktop operating system, starting a word processor within one second is merely desirable, whereas the result of a real-time system in a car that fails to complete its brake subroutine in time, can be catastrophic. However, missing a deadline in real-time systems is not always equivalent to total failure. The result of a task that completes after its deadline might simply become less useful, or irrelevant. When a task must be guaranteed to finish within its deadline, it is called a *hard task*. [4].

2.2 Task Model

A real time system typically consists of a set of tasks, which specify a sequence of computations that are to be executed by the system. Only one task can be executed (running) on a single CPU at a time. We denote a task with priority i as τ_i. Priorities are unique and a lower value of i represents a higher priority. We consider the *periodic task* model, where a *job* is the execution of a single task instance and might in turn consist of multiple *subjobs*. The k^{th} instance of a task τ_i is denoted as $\iota_{i,k}$. Every fixed *period*, T_i, a new job is made available at its *release time* (or *arrival time*), $r_{i,k}$. The execution of a job must complete before its *deadline*. Some time after a job k is released, it is *activated*, the point at which it is scheduled to run (i.e. is assigned the CPU and starts executing). The time at which the job completes its execution, is called *finishing time*, $f_{i,k}$. The response time $R_{i,k}$ of a job is the difference between its finishing time and its release time, $f_{i,k}$ - $r_{i,k}$.

We thus describe a task τ_i by a tuple (ϕ_i, T_i, C_i, D_i), where T_i is a task's period, C_i is the computation time of a single job. ϕ_i is the task's phasing, which is the release time of its first job (i.e $r_{i,0}$) [4]. Finally, D_i is the task's relative deadline: each job instance $\iota_{i,k}$ must finish within D_i time units after its release time $r_{i,k}$. We denote the set of periodic tasks in a system as Γ. Finally, a collection of tasks define an *application*.

2.3 Scheduling

In general, scheduling is the process of deciding which task is granted access to a shared resource at a given time. The CPU scheduler thus decides at which time which task is allowed to run on a shared CPU. In this thesis we consider *fixed-priority scheduling* (FPS), where each task has a fixed priority which is assigned when the task is created. This is in contrast to *dynamic priority*

schedling algorithms like *Earliest Deadline First* (EDF), where a task's priority is changed during runtime, according to the relative deadlines of other tasks.

The scheduler always selects the ready task with the highest priority to run. When the task can be interrupted by another task during its execution, the scheduling mechanism is called *preemptive*. Conversely, the scheduling mechanism is *non-preemptive* if a task cannot be preempted before its job is completed.

2.3.1 FPPS

When tasks are scheduled according to *Fixed Priority Preemptive Scheduling* (FPPS), the *highest priority ready task* (HPT - also denoted as τ_{hp}) is always assigned the processor. If during the execution of the currently running task, denoted asτ_{cur}, a higher priority task τ_j arrives, τ_{cur} is preempted and control of the CPU is assigned to τ_j. Execution of the preempted task is resumed when it becomes the highest priority ready task again (e.g. when τ_j completes its job). Transferring the control of the CPU from one task to another is called a *context switch* (CS).

2.3.2 FPNS

Using *Fixed Priority Non-preemptive Scheduling* (FPNS), no preemptions occur during the execution of a job. If during the execution of a job of τ_i, a higher priority ready task τ_j arrives, it has to wait until execution of τ_i's job completes.

2.3.3 FPDS

Fixed Priority Scheduling with Deferred Preemption (FPDS) [41, 42, 40, 30, 10] only allows preemption at certain specified *preemption points*, and is thus a generalization of FPNS. When scheduling a task τ_i according to FPDS, a job $\iota_{i,k}$ consists of a number of K_i subjobs $\iota_{i,k,1}, .., \iota_{i,k,K_i}$, and preemption points are added between these subjobs (and $K_i \geq 1$). It offers a balance between the coarse grained preemptions of FPNS and the fine grained preemptions of FPPS. Compared to FPPS, which allows arbitrary preemptions, the system overhead using FPDS can be less due to less context switches. Furthermore, in FPDS, points at which preemptions are allowed to happen can be chosen such that they are convenient or efficient for the task. When a preemption point is placed around a critical section, we avoid the need for resource access protocols, as only one task can request the critical section at the same time. Compared to FPNS, it improves the response times and schedulability of higher priority tasks by allowing a finer granularity of preemption points.

To further reduce the overhead of preemptions, *optional preemption points* [30] are introduced. A normal preemption point calls the scheduler to yield its execution to any available higher priority ready task. The disadvantage of this approach is that a task gets no feedback on whether or not it will be preempted. When it would not be preempted, a different execution path could be chosen, such as initiating a longer data transfer. Finally, if the task is aware of the availability of a higher priority ready task, it might prevent the overhead of a system call to a preemption point when it is not needed.

2.4 Servers

A *server* is a special periodic task, originally introduced to service aperiodic requests [4,5]. We consider three different types of fixed-priority servers: the *polling server*, *periodic server* [16, 52], and *deferrable server* [37]. Whereas a periodic task is defined by its computation time C and period T amongst others, a server σ_i is defined by its capacity Θ_i and period Π_i. During runtime, its available budget β_i may vary and is replenished every period to Θ_i. A server can be scheduled similarily to a periodic task. When a server σ_i is running it serves tasks as long as its budget β_i, which is *consumed* at the rate of one per time unit it is running, is larger than zero. When the budget β_i reaches zero, the server is called *depleted*. A server is not limited to service just aperiodic requests, but can also be used to service periodic tasks with a certain execution budget. For the remainder of this thesis we consider servers to service periodic tasks. The mapping of tasks to servers is given by $\gamma(\sigma_i) \subseteq \Gamma$, which defines the set of tasks mapped to server σ_i. A task $\tau_j \in \gamma(\sigma_i)$ can thus only execute if server σ_i's budget $\beta_i > 0$. The exact behavior of each type of server depends on its budget consumption and replenishment rules, which we describe below.

2.4.1 Deferrable Server

The deferrable server is a *bandwidth-preserving server*. A bandwidth-preservering server σ_i does not discard its budget when the workload is *exhausted*, i.e. when there are no more ready tasks in $\gamma(\sigma_i)$. A server can be in one of the states shown in Fig. 2. When a server is created, its initial state is set to *ready* and is eligible to run. A server in the ready state can be dispatched by the scheduler and activated. When a server is activated, its state is changed to *running*. A server in the *running* state is said to be *active*, and in either *ready*, *waiting* or *depleted* state is said to be *inactive*. A change from inactive to active, or vice-versa is accompanied by the server being *switched in* or *switched out*, respectively. An active server can become inactive for the following reasons:

- It can be preempted by a higher priority ready server. The budget is preserved and the server moves into the *ready* state.
- When the server runs out of budget, the server's state is changed to depleted.
- When $\beta_i > 0$ but the workload is exhaused, the budget is preserved and the server is moved into the waiting state.

2.4.2 Periodic Server

Like the deferrable server, the periodic server is bandwidth preserving. Unlike the deferrable server, the workload of a periodic server is never exhausted: a periodic server contains an idling task with a priority lower than any other task $\tau_j \in \gamma(\sigma_i)$. When no other ready tasks are available, the idling task idles the server's budget away, until the budget is depleted or another ready task arrives. The state transition diagram is shown in Figure 3. The state transitions are similar to that of the deferrable server, except that the periodic server cannot reach the waiting state.

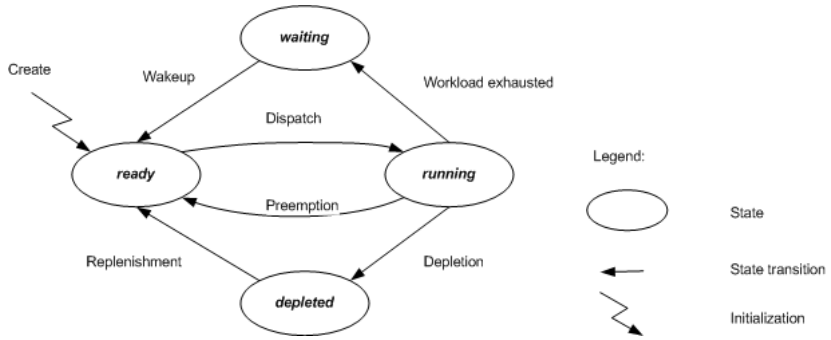


Figure 2: State transition diagram for the deferrable server

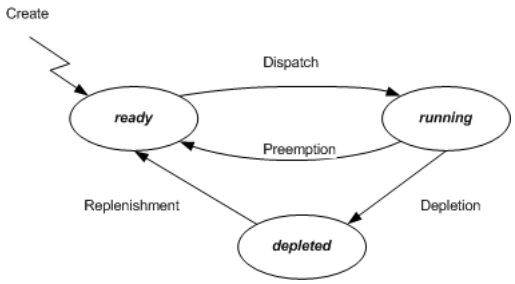


Figure 3: State transition diagram for the periodic server

2.4.3 Polling Server.

Unlike the periodic and deferrable server, the polling server is not bandwidth preserving. As soon as its workload is exhausted, the remaining budget is discarded, and the server is moved into the *depleted* state. Therefore, like the periodic server, it can never reach the waiting state. The state transition diagram is shown in Figure 4.

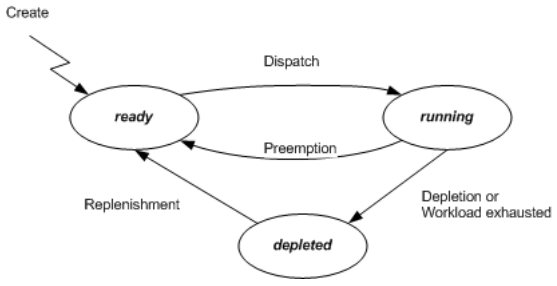


Figure 4: State transistion diagram for polling server

2.5 Resource Reservations

Mercer et al. [8] describe the need for *temporal protection* in real-time systems. When a task under a fixed priority scheduler would misbehave, e.g. by enter-

ing an infinite loop, all lower priority tasks will be preempted and cannot run again. Even when no such errors would happen, traditional real-time schedulability analysis requires the *worst case execution time* (WCET) to be accurately estimated. With the ever increasing complexity of hardware, and dependence on communication with external systems, this becomes increasingly difficult. By introducing processor reservations [2, 8], each task or set of tasks is assigned a virtual part of the CPU, called a processor reserve. This guarantees each reserve a certain amount of processor time.

Rajkumar et al. [1] extended the notion of processor reserves to reservations of other time-multiplexed computing resources as well. They describe a *resource kernel*, which provides protected and timely access to machine resources. In such a system, resources are virtualized so that tasks cannot access a resource directly, but are guaranteed a requested share (or *partition*) of this resource, a *resource reservation*. Various kinds of resources can be virtualized in such a fashion, such as disks, network (bandwidth), processor (CPU time), etc. In order to implement reservations, [1] identifies the following required mechanisms:

- **Admission**: A means to allow applications to specify their resource requirements, and evaluate the reserve requirements to decide whether or not to allow them
- **Scheduling**: schedule the tasks according to their reservations and the admission policy
- **Accounting**: keep track of the execution time used by applications
- **Enforcement**: do not allow reservations to overrun their granted share of a resource.

Resource reservations can thus be used to provide temporal isolation amongst applications, by providing each of them with a virtual share of the platform on which they are running. As the quality and available resources are usually a trade-off, setting resource reserve parameters can also be used for Quality of Service (QoS) mechanisms.

2.5.1 Resource Kernels

In [1], Rajkumar et al. introduce the abstraction of a reservation in the kernel. As described above, this can be a reservation for any time-multiplexed resource. A reservation is defined by the parameters C, computation time, in every period of$T$ time units, within a relative deadline D, with a starting time S, and for a life-time of L time units. When the application requests a reservation with tuple $\{C, T, D, S, L\}$ from the kernel, the request is analyzed with an admission test. The request is granted if the kernel is able to reserve, from time S to $S + L$, C units of the resource in every period of T time units, within a relative deadline of D time units. Depending on the type of the reservation, C can represent actual time (such as for a CPU time reservation), or another unit relevant to the reservation, such as disk blocks, kilobytes of memory or packets of network traffic. In an interval T, when C amount of resources are used up by the reservation, the reservation is *depleted* (versus *undepleted* when the reservation still has budget left). After T time units, the reservations is *replenished*: it gets a new budget of C units of the resource. In this thesis we will consider

hard reservations: when a reservation is depleted, tasks sharing this reservation are suspended and cannot be scheduled until the reservation is replenished. A reservation cannot use any spare resources left in the system that are not used by other reservations (e.g. spare bandwidth for network reservations, or spare processing time for cpu reservations).

2.5.2 Processor Reservations

In this thesis we will look at how to extend an existing real-time operating system with hard processor reservations. From the resource kernel point of view, a processor reservation is defined by its computation time C, which is guaranteed in every period T. This is analogous to a bandwidth-preserving server σ_i with period $\Pi_i = T$ and capacity $\Theta_i = C$. We therefore use the bandwidth-preserving periodic and deferrable server to implement reservations. Just like a server, a processor reservation can be shared by any number of tasks. In order to schedule the servers, as well as the tasks within a server, we use a *hierarchical scheduler framework* (HSF). While admission is performed offline, the other ingredients (see section 2.5) needed to support reservations are thus implemented by servers (accounting and enforcement), and HSF (scheduling).

2.6 Hierarchical Scheduling Framework

Hierarchical scheduling frameworks have been designed to support the integration of independently developed and analyzed *subsystems* [27]. Each subsystem requests an execution budget (share of the CPU time) in wich it can run any number of tasks. This provides each subsystem with a virtual platform and offers temporal isolation to prevent undesirable interference from other subsystems. We use servers to map budgets to subsystems.

A *two-level hierarchical scheduler* allows each subsystem to use its own *local scheduler* to schedule its tasks, using its own scheduling algorithm. Determining which subsystem should be allocated the processor, is in turn done by a *global scheduler* (also called *server scheduler*) [16]. A subsystem is thus defined as a server (with tasks) and a local scheduler.

In the more general case, where subsystems can again contain other subsystems, the hierarchical scheduler may have an arbitrary number of levels. In such a system, the abstraction of resource reservations is recursively applied to all subsystems, such that a subsystem itself supports temporal isolation amongs its sub components. The hierarchy of subsystems and can be represented by a tree-like structure, see Figure 5.

Different schedulers may have different scheduling policies, which allows subsystems that assume different local scheduling policies to be integrated easily into one system. In this thesis we will be focusing on a two-level hierarchical scheduler, where we use a fixed priority scheduler on both local and global level.

2.7 Timing

An important aspect of Real-Time Operating Systems is the ability to satisfy timing constraints. For our extension of μC/OS-II with periodic tasks and reservations, we need to handle time triggered events (*timed events*), such as periodic task arrivals, budgement replenishments and depletions. To this end we need

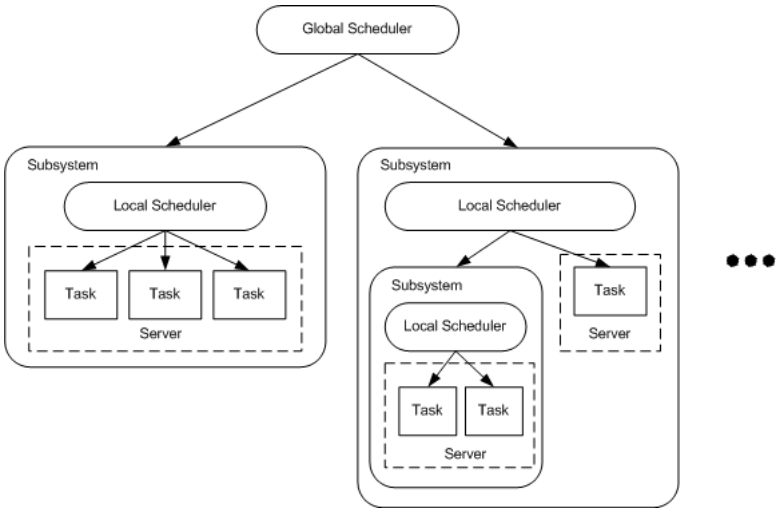


Figure 5: An example of Hierarchical Scheduling

some form of timed event management in μC/OS-II. To prevent an operating system from having to actively request (poll) the current time continuously, timers are used to notify the system at requested times. This notification is most commonly done through an interrupt: when the hardware timer reaches its programmed count, it "fires" an interrupt. Timers can be divided into two types: one-shot timers and periodic timers. Specifying time for the timers can be done in absolute or relative time. The following sections describe the various timing mechanisms and considerations for timer management in real-time operating systems.

2.7.1 Relative and Absolute Time

Time can be expressed in two ways: absolute or relative. A relative timestamp depends on some reference time. The absolute time can then be found by adding the relative time and the reference time together. An absolute timestamp thus already contains the reference time in its value. The reference time itself is commonly (e.g. many Unix-based systems) expressed in terms of seconds since system boot or 1/1/1970 00:00. As time progresses, the absolute time value gets larger, and more bits are needed to store this reference time. Relative timestamps take less space to store: the reference point is only stored once, and not embedded into every timestamp. A timestamp can store a value of $2^n - 1$, depending on the number of bits n. When the maximum value is reached, it wraps around, a problem that occurs more often when using absolute timestamps, as more space is needed to store the timestamps. Waiting until an absolute time is done by comparing the current time to the required timestamp. Waiting until a relative time is usually done by counting (either increasing or decreasing a counter) and comparing the counter to the timestamp. Using relative time saves space, and avoids the problem of having to calculate the exact absolute time for operations such as "delay for 100ms" (by the time T_{now} + $100ms$ has been calculated, T_{now} can have changed). A downside is that all

timestamps expressed in relative time need to be decremented as time progresses to keep their meaning. When using several relative timestamps ordered by time, they can be stored as being relative to each other, only requiring the head timestamp to be updated, a property we use for storing events as described in Chapter 4.

2.7.2 Periodic Timers

A periodic timer is programmed once and instructed to generate a timer interrupt at regular intervals. The resolution of the timer is thus bound by the smallest interval (maximum frequency) the timer supports. The interval of a periodic timer is also called *tick,* and the timer interrupt the *tick interrupt*, handled by a corresponding *tick handler*. A higher frequency means more accuracy, but also leads to more overhead, as the interrupt needs to be handled on every tick. Handling the tick interrupt requires a switch from the task to the interrupt and back, and the execution of the tick handler. Since periodic timers need not be reset after they fire, the overhead of programming timers only occurs at the system's initialization.

In most Operating Systems, a tick handler is used to periodically perform accounting (keep track of the tasks' CPU usage), update the system time, handle due timed events, update the task ready queue and call the scheduler to determine the highest priority ready task. Activating the scheduler periodically with the tick interrupt is also called *tick based scheduling*.

Although using tick interrupts is a convenient way of handling all OS activities at regular intervals, it requires the OS to check if accounting or scheduling needs to performed at every tick. As the tick can also be used for general purpose timing within systems, high frequencies are sometimes used to allow for the resolutions needed for handling timing in video and networking applications, leading to high overhead of tick interrupts and also power usage. The overhead is best noticeable when the CPU is idle, in which case the overhead can be up to 80% for some 1000Hz systems [20]. Therefore, moving away from a (high frequency) periodic tick mechanism is an interesting option in trying to reduce the system overhead.

To remain compatible with the hardware ports and tick-based structure of the μC/OS-II kernel (described in Chapter 3), we take the middle way by keeping the periodic tick, but reducing the computation time of the tick handler. The tick handler only performs work when the next event is due. This brings our system closer to an event based solution, while still keeping a notion of ticks as needed by the μC/OS-II kernel and other applications built on top of μC/OS-II.

2.7.3 One-Shot Timers

A one-shot timer is programmed to fire (generate a timer interrupt) once at a specific time, either counting down or up until a specific timestamp. Once the timer fires, it is deactivated and needs to be reprogrammed. One-shot timers are usually the most flexible in terms of resolution and can thus provide higher accuracy. Periodic timers with high resolutions have too much overhead due to their high interrupt frequency, a problem the one-shot timer does not have. This comes at the price of the overhead of having to reprogram the one-shot

timer every time it fires. With one-shot timers it's not always possible to express expiration times in terms of relative intervals (e.g. "5 ticks from now"). This requires applications to read the current time, and use it to calculate the next absolute expiration time. This, too, might introduce extra overhead and drift. As one-shot timers can be programmed to only fire at the expiration of a next event, the interrupt handler only handles events and calls the scheduler when an event is due. This is also called *event based scheduling.*

A hardware solution called the *High Precision Event Timer* (HPET) [21], solves most problems related to the use of one-shot and periodic timers. It allows to program the timer to fire relative to the last expiration time. This prevents the need for recalculating the absolute time of the next event, avoiding drift (see section 2.7.4). The overhead of programming the timer is also very low, allowing for high timer accuracies and thus preventing jitter.

When events follow up too close on each other, the system spends a lot of time programming one-shot timers and the overhead becomes significant or in the worst case renders the entire system unresponsive. In [20], the use of *smart timers* is suggested, based on *firm timers* [45] in combination with one-shot timers. This concept is based on using an overshoot parameter S. When a timer is requested to expire at time T, the timer will be set to fire at $T + S$. If the kernel is entered at time t, $T \leq t < T + S$ (either due to a system call or some other kernel trap mechanism) the event handler is executed there. In this case, the cost of context switching for an event is completely avoided, as the price of switching to the kernel was already paid by the system call. If the kernel is not entered during this time, the timer will fire at $T + S$, causing an extra interrupt context switch and execute the event. As it enters the kernel to execute the event, costs are saved by also executing all the other events with times up to $T + S$. Setting S to 0 results in pure one-shot timers. Although batching closely follow-up events leads to less timer programming, the use of the overshoot parameter reduces the accuracy (increases jitter), which makes this solution less suitable for real-time systems.

2.7.4 Clock

When using periodic timers, one way for the system to keep track of the current time, is using a tick counter, which is increased at every tick. Some applications, however, require the system time with a higher resolution than that of the periodic tick timer. When no periodic timers are used, another way of reading the current time is also required. One possibility is mapping a hardware clock to a part of the system memory, that can be read by the application. An example is the time stamp counter on the x86 platform [43]. Whether or not a separate hardware clock should be used depends on the hardware availability, the resolution requirements and the overhead of reading the hardware clock. The standard μC/OS-II hardware port has no additional support for a hardware clock, and uses a tick counter to keep track of the time.

2.7.5 Multiplexing

Most real-time systems have to track many different timed events, requiring many timers, with only a limited number of hardware timers available. Therefore, a single hardware timer is used to drive, or *multiplex,* several different

software timers. This reduces the overhead of having to program many different timers, and lowers the hardware requirements. A common way of multiplexing timers is to store a (sorted) queue of software timers with their associated expiration timestamps. In the case of one-shot timers, the timer is programmed to fire at the time of the software timer that should fire first. When handling the first event (and all other events that are due at the same time), the timer is reprogrammed to the time of the second event, and so on. In case of a periodic timer, at every period, the system checks if the current tick count is equal to the first software timer.

2.7.6 Jitter & Drift

Jitter is the deviation between the desired time for an action to be performed, and the actual time it is performed [19]. The bound on this error is defined by the absolute jitter. A system suffers from drift when the absolute jitter is unbounded. Drift in event and timer management can have various causes:

- Inaccurate hardware.
- Event times are not a multiple of the timer granularity. For both one-shot and periodic timers, it not possible to distinguish between events whose time difference is less than the timer granularity. The systems programmer should be aware of the resolution limitations, especially if a hardware abstraction layer automatically rounds the event times to match the available hardware's granularity. Another factor might cause problems when rounding times: the order in which the OS handles timers that expired within a system clock period. For example, an event at time 1.5 and 1.6 both expire between tick 1 and 2. In Operating Systems like Linux and Windows NT4 [5], the timer with the latest experiation time is handled first, which might break temporal constraints. When tasks are scheduled in or out, accounting of tasks is also rounded to the next tick.
- Lengthy timer interrupt service routines.
- Programming a one-shot timer with a time relative to the current time. Everytime the next timestamp needs to be set, the current time must be read and the timer programmed. This process takes some time, which makes the calculated next expiration time inaccurate.

2.7.7 Virtual Timers

On a virtual platform, subsystems often require timed events which are triggered relative to the consumed budget. For example, a timed event should be triggered once the subsystem has been running for 5 time units. As the subsystem could have been switched out, the 5 time units inside the subsystem might have been 10 actual wall clock time units. A budget depletion event is an example of an event that depends on a virtual timer. When no support for virtual timers is present, all virtual event times need to be updated when a subsystem is switched back in.

2.8 Existing Implementations

We looked at several existing implementations of periodic tasks, reservations, hierarchical scheduling and timer management and discuss their details in this section. In 2.8.1, we compare it to the approach we take, and summarize the different properties of the implementations mentioned below.

In μC/OS-II [26], a periodic timer is used to keep track of time, and all timestamps are expressed in ticks relative to the current time, in 16-bits. Each task is assigned a relative timestamp variable, OSTCBDly, which is used when a task needs to sleep or wait for a time-out of some function. During every tick, the global tick counter is increased to keep track of the current time. Every tick, the tick handler also loops through all tasks, and decrements each timestamp variable. When the variable is now 0, the "timer" for this task is due, and the delay or time-out event is handled and the task moved to the ready queue. Finally, the tick handler calls the scheduler and returns control to the scheduled task.

The latest version of μC/OS-II also comes with a timer module extension, which supports the creation of user defined periodic and one-shot software timers. The programmer specifies the mode (one shot or periodic), an initial delay, the period or time out and a callback function. When the timer fires the callback function is called. The time is stored in a 32-bit absolute timestamp (in number of ticks elapsed since the timer module started). The timers are not sorted by time. Keeping a non-sorted list of timers would normally require the timer tick routine to compare all timer timestamps with the current time. To avoid this, μC/OS-II introduces a *timer wheel* and divides the timers in a number of so-called *spokes*, where each spoke contains a non-sorted list of timers, and the spoke number of a timer depends on the timestamp modulo the number of spokes. More spokes means shorter timer lists, but also more memory to store the separate spokes. The timer tick routine is implemented as a separate timer task, which is activated when timer task's semaphore is released. This can be done at any time the software tick should occur, which is not neccesarily equal to a tick from the periodic hardware timer. The timer handlers are executed in the context of the timer task. When the semaphore is released, the context is switched to the timer task. Due to requiring a context switch to and from the timer handler task in addition to still needing the regular tick interrupt, we decided not to use this module.

In RTAI Linux [51], the programmer can specify whether to use one-shot or periodic timers for task activiation, and thus offers a more low-level interface for handling task period events. Depending on the task set, the programmer can choose to trade the additional overhead of programming one-shot timers for the more accurate grannularity of the one-shot timers. RTAI also uses an additional periodic timer to keep track of wall clock time. The scheduler can be activiated by either the periodic timer, or the expiration of a one-shot timer. As Linux runs in the background of the real time extension (RTAI), various timer management mechanisms of Linux co-exist with those of RTAI. For other timed events, other timer mechanisms commonly available in POSIX systems can be used, such as signals.

In [10], Bergsma extends RTAI/Linux with an efficient and maintainable implementation of FPDS. He first adds support to FPNS to the RTAI kernel, after which support for preemption points is added. Finally, by using a shared

memory page between kernel and application, a flag is made available to the application which indicates whether a higher priority ready task is available. This is used to implement optional preemption points.

In [11], the authors describe the extension of VxWorks with periodic tasks and hierarchical scheduling. As the kernel, including the VxWorks scheduler itself, could not be modified, their extension manipulates the global task ready queue instead. Both EDF and FP are supported as a scheduling policy. To implement hierarchical scheduling, periodic servers and periodic server events are introduced. On every server period event, when the current server is preempted, its tasks are removed from the ready queue. The tasks of the server that will now execute are added to the ready queue, and - in case of EDF scheduling - their priorities are updated. To keep track of timed events like budget expiration, replenishment and periodic events, they introduce a timed event manager based on timer event queues (TEQ). These queues contain events sorted by their timestamp, expressed in absolute ticks. This 32-bit value will wrap around approximately every 256 hours. To deal with the wrap around, a special wrapping flag is added to every event in the TEQ. The logic responsible for adding and removing items from the queue deals with detecting wrap-arounds. The stored time is thus not linear (a smaller value can indicate a later event), but rather it is represented in a circular time model. A custom time interrupt handler is introduced to handle the due events. To this end, the VxWare's watchdog library (which in turn depends on VxWork's tick handler) is programmed to call the timer handler at the expiration of the first event in any of the event queues. Other hardware timers could also be used to drive the timer handler. The timer handler handles any due events from the currently active server, and manipulates the ready queue accordingly. When an inactive server is switched in, all events that occured while being inactive are handled, and the budget expiration event is recalculated.

This circular time model also used in [11] was introduced by [22], as the Implicit Circular Timers Overflow Handler (ICTOH), which is an efficient time representation of absolute deadlines in a circular time model. It requires managing the overflow at every time comparison and, as the last bit of the timestamp is used to indicate an overflow, is limited to timing constraints which do not exceed $2^{n-1} - 1$. An implementation of the EDF scheduler for the ERIKA Enterprise kernel [50] based on ICTOH, minizming the time handler overhead, was presented in [23].

The SHaRK real-time system [25] uses timer management based on the open-source library OSLib. Events are stored in a single linked list, and arrival times are expressed in 64-bit POSIX timestamps. When an event is due, an interrupt is generated by the hardware timer, which can either be a one-shot or periodic timer. A separate timer is created for each event, e.g. period arrival events, budget events, etc. Periodic tasks are supported by an API to sleep until the next period.

Oikawa et al [28], describe the design and implementation of Linux/RK, an implementation of a resource kernel (Portable RK) within the Linux kernel. They minimize the modifications to the Linux kernel by introducing a small number of call back hooks for identifying context switches, with the remainder of the implementation residing in an independent kernel module. Linux/RK introduces the notion of a resource set, which is a set of processor reservations. Once a resource set is created, one or more processes can be attached to it

to share its reservations. Although reservations are periodic, periodic tasks inside reservations are not supported. A task can simply wait until its parent reservation's period, by blocking on a reservation replenishment event. The system employs a replenishment timer for each processor reservation, and a global enforcement timer which expires when the currently running reservation runs out of budget. Whenever a reservation is switched in the enforcement timer is set to its remaining budget. Whenever a reservation is switched out, the enforcement timer is cancelled, and the remaining budget is recalculated. Each timer consists of a 64-bit absolute timestamp and a 32-bit overflow counter. The timers are stored in a sorted linked list. When a timer interrupt occurs, which is driven by an additional (i.e non-Linux) one-shot timer, the timer handler checks for any expiring timers, and performs the actual enforcement, replenishment, and priority adjustments. For monitoring purposes, the status about reserves is available to other applications through the Linux process-file system (*proc fs*). An extension of the Linux/RK kernel with hierarchical reservations is described in [17], where each (sub-)reserve has its own replenishment timer.

In [46], Kato et al propose a loadable real-time scheduler framework for Linux, called RESCH. It aims to provide easy installation of multicore scheduling algorithms into Linux through plugin modules, without the need to patch the kernel. Processor reservations are provided through servers, which can contain one task. Enforcement and accounting is done similarily to the method used by [28], except Linux' internal timer mechanisms are used. Temporal granularity and latency thus also depend on the underlying Linux kernel. Periodic tasks are supported, and a function is provided to wait for the next period, implemented by using a Linux sleep function. AQuoSA [48] also provides the Linux kernel with EDF scheduling, in combination with various well-known resource reservation mechanisms, such as the constant bandwidth server [4]. Like [28] it requires a kernel patch to provide for scheduling hooks, and the implementation of the timers is similar to that of [46]. CPU reservations are provided as servers, where a server can contain one or more tasks. Periodic tasks are supported by providing an API to sleep until the next period.

In [49], Eswaran et al. describe Nano-RK, a reservation-based RTOS targeted for use in resource-constrained wireless sensor networks. It supports fixed priority preemptive multitasking, as well as resource reservations for cpu, network, sensor and energy. Only one task can be assigned to each cpu reservation. Nano-RK also provides explicit support for periodic tasks, where a task can wait for its next period. Timer implementation is based on the 64-bit POSIX time structure. The authors assume the timestamp value is large enough that it will practically not overflow. Each task contains timestamp values on its next period, next wakeup time and remaining budget. A one shot timer drives the timer tick, which calls the scheduler. It loops through all tasks, updates the tasks' accounting and determines the next wake up time. The next tick handler is invoked on the first next wake up event.

2.8.1 Comparison

Unlike the work presented [46,47,48,49,51] which are based on Linux, we focus on extending an RTOS targeted at embedded systems with limited resources. Our design aims at efficiency in terms of memory and processor overheads, while minimizing the modifications of the underlying RTOS. Like [25,28,11] we use

event queues to prevent having to loop through all tasks in the timer handler to determine due events. Our RELTEQ approach (see Chapter 5) provides long event interarrival time of the event queues, while keeping overhead low. We use a relative time model to prevent drift, prevent the need for overflow checking and use less memory to store the timestamps. Unlike all the other works, RELTEQ also supports efficient virtual events, which avoids the need of having to cancel, recalculate and reinsert virtual timers such as budget depletions. Like [11], we also avoid interference from handling timed events of other subsystems, but extend this mechanism to also work with deferrable servers. Qualitatively, RELTEQ provides versatile and concise central timer management. It is interesting to note that many works provide no central module for general timer management. Different modules take care of different types of timing aspects, which can lead to code duplication and additional overhead. This is mostly due to the fact that these works are built on top of larger existing OSes, where it would not have been practical (or possible, in the case of closed commercial systems) to modify the existing architecture. Combined with the lack of virtual timers, this means accounting and updating timestamps need to be handled throughout different functions, whereas events in our design are handled in a central place (event handlers), making the code more maintainable. Table 1 and 2 summarize the time models and features of the works discussed above.

Model	I	T	overflow handling	n	V
Linear ([25,28,46, 47, 48, 49,51])	2^n	A	not supported	64	N
μC/OS-II	2^n	R	not supported	16	N
Circular [23,P11]	2^{n-1}	A	stored in last bit	32	N
RELTEQ	∞	*R*	*dummy events*	*16*	*Y*

Table 1: Time model comparison. I is the maximum interval between any two events in the queue, n is the number of bits used to represent timestamps. V is support for virtual events (N: No, Y: Yes). T stands for time representation (A: absolute, R: relative).

Implementation	Timer Management	Kernel	HS	FPDS	PR	SI	Servers	TH	P
[28,46,47,48,49]	Linux[1]	Linux	2[2]	N	Y	No	Y[3]	Varies[1]	Y
RTAI Extension [10]	Linux[1]	Linux	N	Y	N	N/A	N/A	Varies[1]	Y
SHaRK	OSLib (Event Queue)	SHaRK	N[7]	N	Y	No	Y[3]	Due events	Y
VxWorks+HSF [11]	VxWorks, TEQ	VxWorks	2	N	Y	Yes	PS[6]	Due events	Y
Linux/RK [28]	Event Queue (+ Linux)	Linux	n[8]	N	Y	No	PS	Due events	Y[5]
NanoRK [49]	Task-based period, wakeup, budget	NanoRK	N	N	Y[4]	No	PS	All tasks	Y
μC/OS-II	Task-based delays & time-outs	μC/OS-II	N	N	N	N/A	N/A	All tasks	N
Our implementation	*RELTEQ*	*μC/OS-II*	*2*	*Y*	*Y*	*Yes*	*PS,DS*	*Due events*	*Y*

Overview of related work implementations. Abbreviations: HS = levels of hierarchical scheduling supported, N = not supported, Y = supported, N/A = Not Applicable, PR = Processor Reservations, SI = avoids interference of inactive subsystem events. PS = Periodic Server, DS = Deferrable Server, TH = inspected by Timer Handler, P = kernel primitives for periodic tasks provided.

[1] Various Linux timers (High resolution, Jiffy-based), different timer handlers.

[2] If servers are provided, see [3].

[3] Various server algorithms provided by plug-ins in different works, e.g. [48] implements servers such as CBS and more.

[4] Multi-resource reservations (power, cpu, etc.).

[5] By waiting for period of parent reserve.

[6] Also allows EDF scheduling, we focus on FP, but can be extended easily as done in [44].

[7] No mention found in documents or source code, but might exist as a plug-in.

[8] An extension to provide for reservation hierarchies is described by [17]. No subsystem schedulers.

Table 2: Implementations

3 μC/OS-II & OpenRISC 1000

μC/OS-II is a real-time preemptive multitasking operating system for embedded systems, maintained and supported by Micrium [26]. It is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Both source-code and documentation are provided by Micrium. It is written in highly portable ANSI C, and as such has been ported to various different hardware platforms.

3.1 Overview

μC/OS-II kernel services are split up in different modules (.C files, with a shared header file UCOS_II.H). Through these modules, μC/OS-II provides services like semaphores, mutices, mailboxes, queues, memory & task management. Modules can be enabled or disabled at compilation time, using compiler directives (*#define* in C). This allows programmers to easily link a custom version of the kernel to their applications, with a minimal memory footprint.

The kernel itself is platform independent and depends on functions provided by a hardware platform-specific *port*. The port consists of three separate parts to implement context switching (between tasks and tasks or tasks and interrupts), program the periodic tick interrupt, enable/disable interrupts, and configure the hardware (e.g. stack growth direction, clock frequency, etc.). Certain hooks are also provided in the μC/OS-II kernel that can be used by the port to specify additional actions when a task is created or a tick interrupt occurs.

An overview of the architecture is shown in Figure 6. The overview is shown in the form of a component diagram [6], where a component can consist of multiple *modules* (also called *blocks*). When the component icon is shown in the right upper corner of a square, the modules making up the component are shown.

A mapping from modules to source code files can be found in Appendix B. For the architectural diagrams in the remainder of this thesis we leave out the hardware port and configuration option details, and consider μC/OS-II to refer to the μC/OS-II Kernel. As we focus on the architecture of our extensions later on, we also leave out the explicit mention of the interface between μC/OS-II and the application.

3.2 Memory model

The application - in one or several separate source files - is compiled against the rest of the kernel. Once linked, the kernel, port and application thus form one image. Depending on the eventual hardware platform, this image is burnt into a ROM or stored in a RAM or other memory. Unlike most general purpose Operating Systems, tasks are not separate programs that are "loaded" from e.g. a disk, but are static and part of the final image. Similarly, memory for the data structures, e.g. the table of tasks and events, is statically allocated and stored inside the image. The kernel has no explicit support for different memory segments or priviledge levels. In both the default x86 (Intel) port and the OpenRISC 1000 port [29], the kernel and application tasks share a single memory space. Although this means there is no memory protection, no

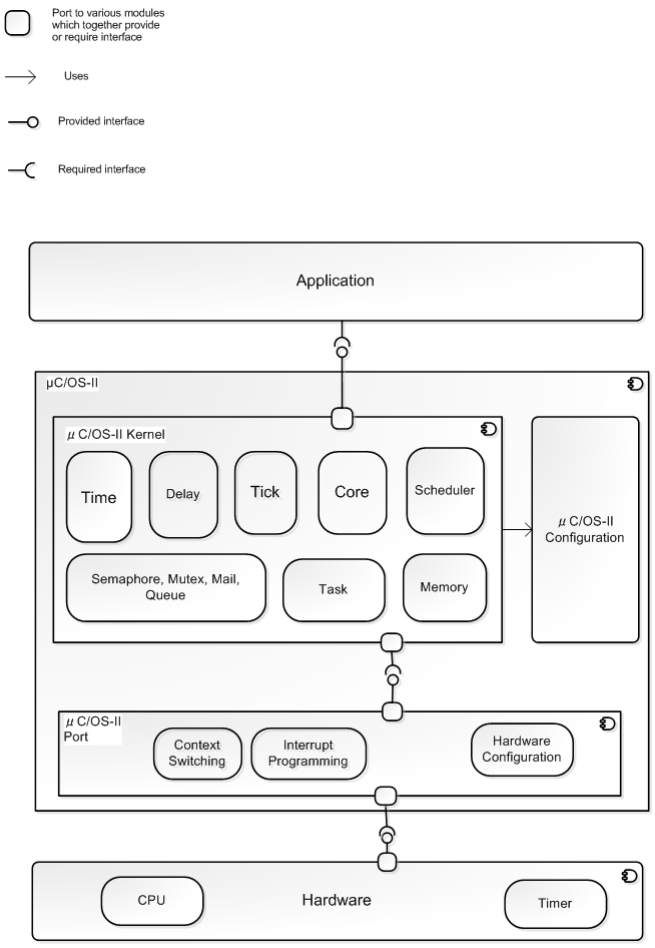


Figure 6: Overview ofμC/OS-II' architecture.

additional virtual memory management is needed and applications can directly call the kernel API primitives, without the need for costly system calls to switch to kernel protection mode.

3.3 Timer Interrupt

Although μC/OS-II has support for servicing multiple and nested hardware interrupts, only a timer interrupt is enabled by default. Other platform-specific interrupts can be defined in the port module. The timer interrupt is a periodic interrupt, and is programmed at a certain frequency, typically between 10 and 100Hz. Each timer interrupt is called a *tick*. The tick interrupt handler updates the system time and the timer count for each task, which are used to keep track of delays and time-outs for synchronization primitives. At the end of each interrupt handler invocation, *OSIntExit()* is called, which returns the control to the task (if the interrupt was nested, control is returned to the previous interrupt handler). Before returning from the outermost interrupt handler, the scheduler

is called to determine the highest priority ready task and returns control to that task by loading (restoring) its context. Figure 7 describes the process of a timer interrupt in pseudo code, where τ_{hp} is the highest priority ready task, and τ_{cur} is the currently running (interrupted) task. *ContextSwitch*(τ_{cur},τ_{hp}) switches the context from the currently running task τ_{cur} to τ_{hp}, and sets τ_{cur} to τ_{hp}.

```
TimerInterrupt():
  OSIntEnter()
  TickHandler()
  OSIntExit()

OSIntExit():
  InterruptNesting = InterruptNesting − 1
  if InterruptNesting = 0:
    τhp = Scheduler()
    if τhp ≠ τcur:
    context = ContextSwitch(τcur,τhp)
  RestoreContext(context)

OSIntEnter():
  context = SaveContext()
  InterruptNesting = InterruptNesting + 1
```

Figure 7: Pseudo code for *OSIntExit()*, *OSIntEnter()* and *TimerInterrupt()*

3.4 Task Model & Scheduling

μC/OS-II supports 64 tasks by default, and can be extended to 256 by changing the configuration settings. Each task is described by a Task Control Block (TCB). This structure contains information about the task's priority, name, stack, and state. As the task's priority is also used as a unique identifier for the task, only a single task can be assigned to each priority-level. The TCB also contains pointers to the next and previous TCB in the list. A TCB can be part of either the *free TCBs* list, or *tasks* list, and the next and previous pointers are used to traverse either list. When a task is created, the TCB for its priority is added to the *tasks list*, and removed from the free list. During this project, we have not considered the extension to 256 tasks, and assume a maximum of 64 tasks.

A task can be in one of the states shown in Figure 8. As all 64 TCB structures are statically allocated, the state in each TCB is initially set to the *dormant* state. Once a task is created, the corresponding TCB is added to the *tasks list*, and its state is set to *ready*. A task in the ready state can be dispatched by the scheduler, at which point the task is moved into the *running* state. When the task is blocked (e.g sleeping or waiting for a resource or synchronization primitive), it is moved into the *waiting* state. From this state, a task can be woken up again by the tick handler (which handles wake-ups from sleeps), or by another task which releases a resource the task was waiting for. A task which is in the running state is said to be active, and inactive otherwise (dormant, ready, waiting).

The system keeps a ready task table, *OSRdyTbl*, which has 8 8-bit entries,

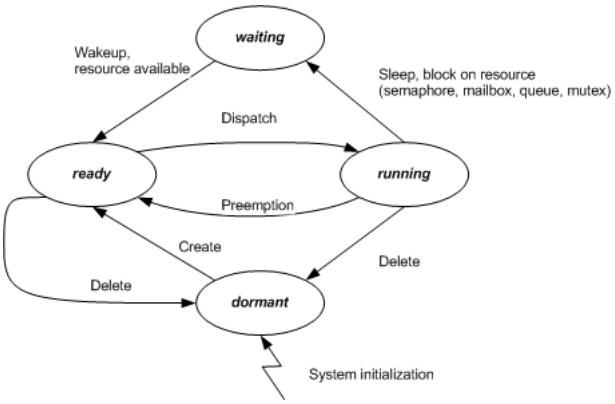


Figure 8: Task states in μC/OS-II

called a *group*. Every bit in one of these groups represents the ready state of a single task, and a group thus contains 8 tasks. The lower the number, the higher the priority, i.e. priority 0 at index 0 in the table contains the ready state of the highest priority task. A separate 8-bit variable *OSRdyGrp* keeps track of which of the 8 groups contain ready tasks (i.e. every bit corresponds to one group). During run-time, whenever a task becomes ready, its bit in the ready table and its group bit in *OSRdyGrp* is set. Inversly, when a task becomes non-ready (*waiting* or *dormant*), its ready bit is unset and if the group byte is 0 (meaning the group contains no more ready tasks), the group bit is unset.

The scheduler is responsible for determining the HPT. Once the scheduler is called, and the returned HPT is different from the current task, a context switch is performed. The scheduler is implemented by the function *OS_SchedNew()*. To find the HPT, the scheduler simply selects the highest bit from the ready table entry given by the highest bit in *OSRdyGrp*. μC/OS-II stores a look-up table, *OSUnmapTbl*, to map an 8-bit bitmask value to the highest bit value. For example, if OSRdyGrp is 00010100 (decimal value 20), the highest priority group that contains a ready task is 2 (the third group, as the third least significant bit is 1). The complexity of the scheduler is therefore $O(1)$ and not proportional to the number of tasks created in the system.

3.5 OpenRISC 1000 port

All simulations and measurements were run using the OpenRISC 1000 simulator [29]. It is maintained by OpenCores, which provides it as a free and open source simulator of the OpenRISC 1000 architecture. It also includes a Software Development Kit (SDK) based on the GNU tools (compiler, linker and debugger). The simulator is cycle-accurate, and the timing behavior is independent from the load on the host operating system, which makes it suitable to perform measurements on a real-time OS. Within our group we ported the latest version of μC/OS-II (v2.86) to OpenRISC 1000 [35], based on the port by Oliver [37].

3.6 Measurements

With all our measurements in this thesis, the OpenRISC 1000 simulator was running at 100Mhz. The tick frequency was set to 100Hz, i.e. each tick takes 10ms. At 100Mhz, every cycle corresponds to 0.01μs (10ns), and every tick (at 100Hz) is equal to 1,000,000 cycles.

The interface to the simulator is based on console output only. In order to perform measurements we use the NOP instruction of the OpenRISC 1000 architecture, through which the simulator provides profiling to the developer by dumping the current cycle and instruction count to the standard output. Although the host OS needs to perform the actual printing of the cycle count to the console, this does not take time inside the simulator. Therefore, the only overhead of the profiling point is the single NOP instruction itself (which does not perform any other operation). Unfortunately, no additional debug text or variables can be provided to the NOP function. As the console will simply be filled with a list of cycle count dumps, this makes it hard to identify which profiling point a number matches to.

To resolve this problem, we developed a separate profiling extension, which - like the other μC/OS-II modules, can be enabled and disabled in the configuration. It provides a single function: *profilePoint*(t, n), where t is some text and n some number which can be used to identify the point. When the function is called, it stores the text, the number and the current system time (tick count) in a global table of profiling points. The maximum number of profiling points is configurable. When the application ends - and any overhead is no longer important - the list of profiling table entries is printed to the standard output. As the *profilePoint* function only stores information into an array, the overhead is low enough not to disrupt the program flow, but too significant to be included in the measurements. That's why the *profilePoint* actually calls the NOP instruction *twice*. When a profile point is reached, the first cycle count is dumped immediately. Then, the profile point information is stored into the global table. When the point has been stored, the second cycle count is dumped. This way, the overhead caused by the point itself can be substracted. The second part of the profiling system (which runs in the host OS) now captures the standard output from the console, and matches the $2n-1$th, and $2n$th cycle count output to the nth table entry line output.

Console output

```
100
120
300
320
400
420
600
620
700
720
start task, -, 1
start job, 1,3
end job, 1,4
start job, 2, 6
end job, 2, 7
```

(a)

Program Code

```
Task:
        profilePoint("start task")
        jobs = [1,2]
        for job in jobs:
                profilePoint("start job", job)
                calculation()
                profilePoint("end job", job)
                job = job + 1
```

(b)

Measurement results

#	Tick	Name	Var	Begin	End	Delta
1	1	start task	-	100	120	-
2	3	start job	1	300	320	180
3	4	end job	1	400	420	80
4	6	start job	2	600	620	180
5	7	end job	2	700	720	80

(c)

Figure 9: Example measurements using the profiler. (a) shows the output it generates at run-time to the console, (b) shows an example task with profiling points, (c) shows the table of measurement points which is constructed by matching the cycle counts and table output lines. *Name* and *Var* correspond to t and n in function $profilePoint(t, n)$.

Figure 9 shows an example of using the profiler. For this example, the values were made up, and not actually measured. The figure shows the overhead of the profiling point itself is 20 cycles (the difference between '*begin*' and '*end*'). *Delta* is the time between two profiling points, and is calculated by substracting the previous *end* cycle value from the current *begin* cycle value.

All of our measurements were performed using this profiling component. To visualize the execution trace of the tasks, we use the Grasp visualizer, developed by Holenderski et al. [31].

4 FPDS

The current scheduler in μC/OS-II supports FPPS (see section 2.3.1), by which a task can be preempted at arbitrary points. One of our goals is to extend μC/OS-II with FPDS (see section 2.3.3), to reduce the cost of context switches by only allowing task preemption at convenient times. We want to provide the following functionality:

- The application should be able to register tasks to be scheduled according to FPDS. After a task is registered, it can no longer be preempted at arbitrary points.
- An FPDS task should be able to specify a preemption point, at which the task can be preempted, by calling the preemption point procedure.
- An FPDS task can specify an optional preemption point. This optional preemption point procedure returns information to the task about the availability of a higher priority ready task, such that the task may decide to alter its execution path accordingly.

The interface to provide this functionality is specified in the next section.

4.1 Interface

For the remainder of this thesis, when specifying interfaces, we make the distinction between *procedures*, which can change the global state, and *functions*, which cannot. The type of all parameters to procedures and functions are *in-parameters*. When no return value is explicitly mentioned, the function or procedure returns *True* on success, and an error code on failure. When a pre-condition is specified and does not hold when calling the function or procedure, an error code should be returned. A dot-notation is used to denote attributes of an object, e.g. $\tau_i.SchedulePolicy$ refers to the $SchedulePolicy$ attribute of task τ_i.

The default setting of newly created tasks in the system is to be scheduled according to FPPS. Either the task itself, or another entity that for example created the task, can *register* a task to be scheduled according to FPDS. A flag $\tau_i.SchedulePolicy$ indicates whether the task is being scheduled according to *FPDS* or *FPPS*. A task specifies a preemption point by calling the preemption point procedure. Depending on the argument passed to this procedure, the preemption point is optional or not. The provided FPDS functionality is similar to that offered by Bergsma in [10].

SetFPDS(τ_i) - Procedure to register a task τ_i to be scheduled according to FPDS. This stops a task from being preempted by the scheduler, until it calls the preemption point procedure. Pre-condition: a task exists at priority i. Post-condition: $\tau_i.SchedulePolicy = FPDS$.

SetFPPS(τ_i) - Procedure to return to scheduling task τ_i according to FPPS. Pre-condition: a task exists at priority i. Post-condition: $\tau_i.SchedulePolicy = FPPS$.

PreemptionPoint(*allow_yield****)*** - Procedure to specify a preemption point. When the flag *allow_yield* is *True*, the procedure should indeed yield control when a higher priority ready task is available. Setting the argument to *False* can thus be used to insert an optional preemption point. It returns whether or not the procedure yielded (or would have yielded if *allow_yield* is *False*). Pre-condition: $\tau_{cur}.SchedulePolicy$ = FPDS.

4.2 Design & Implementation

As FPDS is a generalization of FPNS, we take a similar approach as [10] and look at both non-preemptive task support and adding preemption points.

Consider a periodic task τ_i, where each job $\iota_{i,k}$ consists of a number of K_i subjobs $\iota_{i,k,1}, .., \iota_{i,k,K_{i,k}}$, and $K_i \geq 1$. Between these subjobs we introduce a preemption point where the system is allowed to switch to a higher priority ready task. Figure 10 shows the general approach for a job $\iota_{i,k}$ of such a task τ_i, where k, j are the instance numbers of the job and subjob, respectively. The actual work of the subjob is represented by the procedure $f_{i,j}(k)$. *SetFPDS(τ_i)* informs the kernel this task cannot be preempted. The *PreemptionPoint()* procedure yields control to a higher priority ready task, if available.

```
ι_i(k):
f_{i,1}(k)
for j in [2..K_i]:
  PreemptionPoint()
  f_{i,j}(k)
end for

Main():
SetFPDS(τ_i)
```

Figure 10: A generic implementation of a FPDS task τ_i, with job $\iota_{i,k}$ split up into K_i subjobs

One way to make a task non-preemptive is by using the functions the μC/OS-II kernel provides to enable and disable the scheduler. These primitives are named OSSchedLock() and OSSchedUnlock() [26]. After the scheduler is disabled, the running task will no longer be preempted, except by interrupts. As OSSchedUnlock re-enables preemption and also calls the scheduler, this primitive can be used to implement a preemption point. Whenever we call the function, and a higher priority task is available, the running FPDS task will be scheduled out. Figure 11 shows an implementation using these functions.

```
ι_i(k):
for j in [1..K_i]:
  OSSchedLock()
  f_i(k, j)
  OSSchedUnlock()
end for
```

Figure 11: FPDS task implementation using μC/OS-II primitives.

The disadvantage of this implementation is that we add non-preemptive parts to an otherwise preemptive task, rather than adding preemption points to a non-preemptive task. The implementation in figure 11 has different points in the code where preemption could occur, namely everywhere outside the *OSSchedLock()-OSSchedUnlock()* block. Depending on the code that is executed outside of this non-preemptive block, and the clock granularity (tick frequency), this could lead to additional context switches. Furthermore, we would also like to introduce optional preemption points (see section 2.3.3). As OSSchedUnlock provides no information about whether or not we will be scheduled out, another mechanism is required.

We therefore introduce several new OS primitives and add an extension to the scheduler, which is explained in the sections below. In order for the μC/OS-II scheduler to call our FPDS scheduler, we modified the original scheduler to provide a hook through which our scheduler is called. An overview of the architecture of the FPDS implementation inside μC/OS-II is shown in Figure 12.

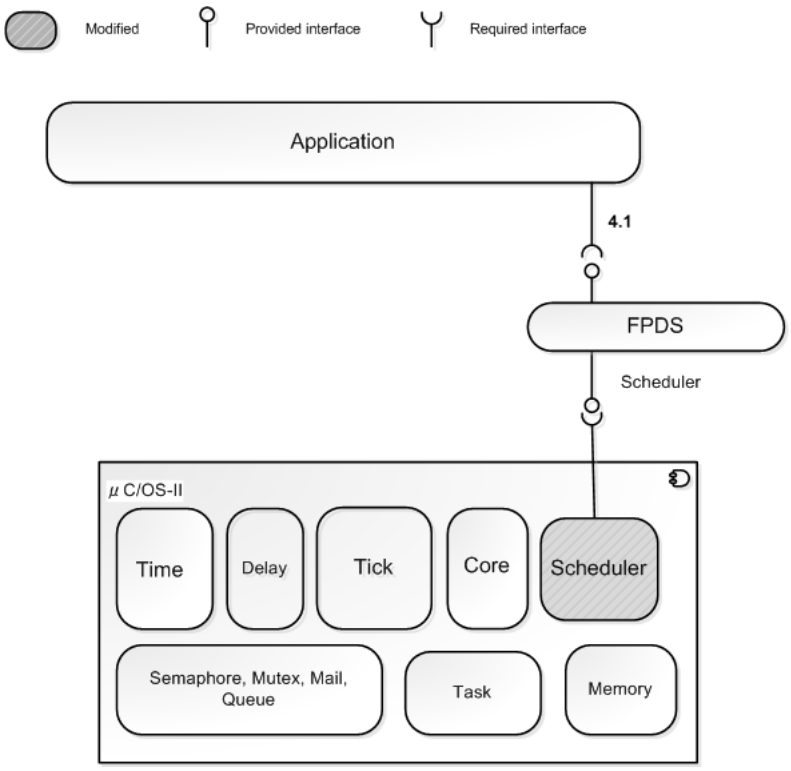


Figure 12: Architecture of FPDS. The number (4.1.) points to the section where the interface is described.

4.2.1 Disabling preemptions

In [10], a task can be set to be non-preemptable using a newly introduced primitive, RT_SET_PREEMPTIBLE($False$). From that point on, the task will

continue to execute until it calls the preemption point procedure, or moves out of the ready state (e.g. by waiting for the arrival time of the next job). We take a similar approach by introducing a procedure, *SetFPDS(τ_i)*, which a task can use to indicate it wants to be scheduled according to FPDS. This simply sets a flag, *SchedulePolicy*, in the task's TCB, which the scheduler checks for.

4.2.2 Scheduler

Whenever the μC/OS-II scheduler runs, it returns the highest priority ready task available. In order to support FPDS, we modified the scheduler to return immediately when the current task's *SchedulePolicy* is equal to *FPDS*, and the task's state is still *running*. When the scheduler returns immediately, it skips the selection of a new high priority task, and therefore no context switch will occur. When the scheduling was skipped, a global flag *ShouldSchedule* is set. This flag is later used by the preemption point procedure *PreemptionPoint()* to check if it needs to execute the scheduler before deciding whether or not it should yield. Whenever the scheduler is fully executed (i.e. task selection is not skipped), the flag is reset. This is different from the implementation in [10], where the scheduler itself is still fully executed, but only any subsequent context switches are skipped while the task is in FPDS mode. When using FPDS, our design thus saves on both context switching and scheduler overhead (including global scheduling which we introduce in Chapter 7). This is possible because μC/OS-II does not depend on the scheduler to keep the ready task list up-to-date, as the ready state is set directly by the OS primitive responsible for the change in ready state. The behavior of the modified scheduler is illustrated by the pseudo-code in figure 13, where τ_{cur} is the current task, and *OriginalScheduler()* is the original μC/OS-II scheduling method.

Scheduler():
if τ_{cur}.*SchedulePolicy* **is** *FPDS* **and** τ_{cur}.*state* **is** *running*:
  FpdsShouldSchedule = *True*
  return τ_{cur}
else:
  FpdsShouldSchedule = *False*
  return *OriginalScheduler()*

Figure 13: Pseudo code for FPDS scheduler

4.2.3 Preemption Point

An important aspect of the FPDS implementation is the overhead of the preemption point. As FPDS aims to reduce the overhead of context switching, the overhead of the preemption point itself should not negate this improvement. When a job arrives at a preemption point, a kernel function is called that yields the control to any available higher priority task. As identified in [30], this mechanism has two drawbacks. First, calling the kernel function can be expensive when the task is running in a different memory space and a *mode switch* (usually through a system call) is required. Such a call is costly when the kernel function finds out that no higher priority task is ready to run, and thus the call would not have been needed. Secondly, the job does not know if it will be preempted

or not. If this information was available, it might choose a different excution path to make better use of longer non-preemptive periods. To remedy these issues, [10, 30] suggest keeping a flag, *ShouldYield*, in the memory shared by the application and the kernel, which the scheduler sets whenever a higher priority task arrives during the execution of the non-preemptive subjob. Whenever a preemption point is reached, this flag can be inspected without having to perform a system call. Furthermore, this flag indicates if a preemption will happen when the job yields, and thus can be used to make decisions about altering the execution path.

As the tasks running in μC/OS-II run in the same memory space as the kernel, no system call is required and the overhead of calling the preemption point procedure is very small. Because we disable the scheduler completely, and do not have to prevent unnecessary system calls, we do not introduce the *ShouldYield* flag mentioned in the previous paragraph. Instead, our preemption point implementation looks at the *ShouldSchedule* flag. When this flag is not set, the scheduler has never been called, which means no new task can have arrived. In this case, we do not have to call the scheduler. When the flag is set, a higher priority task *might* be ready (the scheduler might also have been called because of the arrival of a lower priority task). We thus run the scheduler to establish, if - and which - higher priority task is ready. The result is stored in τ_{hp}, and the *ShouldSchedule* flag is reset. If such a task τ_{hp}, with $\tau_{hp} \neq \tau_{cur}$ is found, a separate argument, *allow_yield*, to the preemption point procedure indicates whether it should indeed preempt the current job, or return. The return value indicates whether the job would have been preempted. This allows the task to use optional preemption points. The pseudo-code in Figure 14 shows the behavior of the preemption point procedure, where *ContextSwitch*(τ_{from},τ_{to}) is used to context switch from the currently running task τ_{cur} to the HPT, τ_{hp}.

PreemptionPoint(allow_yield):
if *ShouldSchedule* **is** *True*:
  ShouldSchedule = *False*
  τ_{hp} = *OriginalScheduler()*
end if
if $\tau_{hp} \neq \tau_{cur}$:
  if *allow_yield* **is** *True*:
    ContextSwitch(τ_{cur},τ_{hp})
  end if
  return *True*
end if
return *False*

Figure 14: Pseudo code for FPDS preemption point.

4.3 Evaluation

4.3.1 Behavior

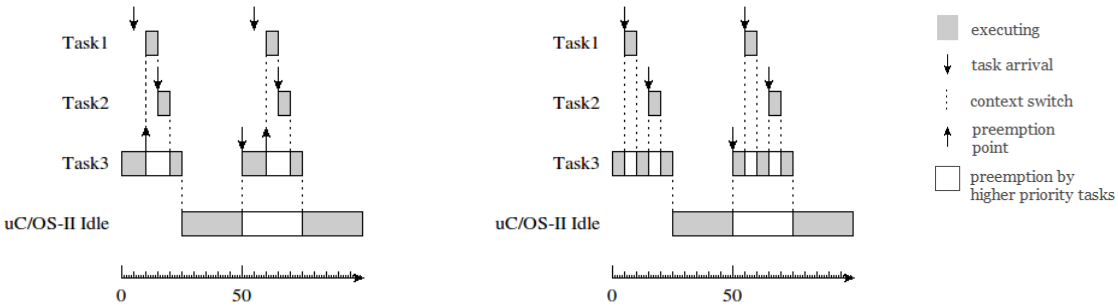
To demonstrate the behavior of the implementation of FPDS scheduling, we created a test task set with the parameters as shown in table 3. Task τ_1 and τ_2

both have one subjob, whereas τ_3 has two subjobs.

Priority	C_i	T_i $(=D_i)$	ϕ_i	$C_{i,k,0}$	$C_{i,k,1}$
1	5	50	5	5	-
2	5	50	15	5	-
3	15	50	0	10	5

Table 3: Task parameters of FPDS test task set.

The task bodies follow the structure as described by Figure 10, where the computation time of $f_i(k, j)$ is given by $C_{i,k,j}$ ticks. The execution trace of running this task set on the OpenRISC 1000 simulator for 100 ticks is shown in Figure 15. In Figure 15a, the task set was scheduled according to FPDS, and FPPS in 15b. In 15a, even though a higher priority task, τ_1, arrives at $t = 5$, task τ_3 is only preempted at $t = 10$, where the first subjob is finished, and thus arrives at a preemption point. In total, 9 context switches occur using FPDS in 15a, where 11 are required in 15b using FPPS.



(a) (b)

Figure 15: Execution trace (a) FPDS, (b) FPPS

4.3.2 Processor Overhead

To measure the overhead of context switches and the FPDS component, a number of profiling points were added to the kernel. We measured the worst case execution time (WCET) of the following parts: context switch (C_{ctx}), the scheduler when in FPDS mode $C_{sched_{fpds}}$ (i.e. when *OriginalScheduler()* is skipped), normal execution of the scheduler (C_{sched}), and the (optional) preemption point C_{yield}, without the scheduling and context switching. The WCET of a preemption point, including the context switch and scheduling is $C_{yield} + C_{ctx} + C_{sched}$. See Table 4.

	C_{ctx}	$C_{sched_{fpds}}$	C_{sched}	C_{yield}
Cycles	1822	190	352	262
μs	18.2	1.9	3.52	2.6

Table 4: Measured overhead of FPDS.

All of the measured values are fairly small, up to the point where declaring another variables on the stack is already significant. This is because the functions mainly consist of simple if statements, and use no higher-level OS primitives.

Our measured value for C_{ctx} is the cost for switching the context (e.g. registers, stack pointer and instruction pointer) between two tasks on the OpenRISC 1000 simulator, without taking into account caching, pipelining, or any DMA transfers that might have been interrupted. In [30] the authors claim these factors are significant for the context switching costs on platforms like the one introduced in Chapter 1. The actual cost of C_{ctx} on such a platform is thus variable. By using FPDS we can lower the context switching cost by placing the preemption points at convenient points.

For as long as an FPDS task is running, and no new task arrives, the FPDS task does not have to relinquish its control over the processor. Whenever a new task does arrive, the scheduler is called, but skipped when in FPDS mode. To yield in a preemption point, we thus always need one scheduler invocation and one context switch. When a new higher priority task arrives during an FPDS task, and the task yields at the end of its subjob, the cost is $C_{sched_{fpds}} + C_{yield} + C_{sched} + C_{ctx}$ - C_{ctxr} = 2626 - C_{ctxr} cycles, where C_{ctxr} is the reduction of the context switch cost (i.e. due to preempting at a convenient point). Under FPPS, when a new task arrives, the cost is $C_{sched} + C_{ctx}$ (2174 cycles).

When using FPDS, we can also reduce the number of preemptions that occur during the execution of a task, as shown in Figure 15. In Fig. 15, at time t = 5, a higher priority task τ_1 arrives. Using FPPS (Fig. 15.b), τ_1 preempts the current job of task τ_3, at a cost of $(C_{sched} + C_{ctx}) * 2 = 4348$ cycles. The context switch is prevented in Fig. 15.a, and the cost under FPDS is again $2626 - C_{ctxr}$ cycles.

4.3.3 Modularity & Memory footprint

The FPDS component is implemented in a separate file, OS_TEV_F.C. It consists of 80 lines of code (LOC). In this thesis, the number of lines of code include compiler directives (configuration switches), but exclude white lines, comments, debug, and profiling code. The μC/OS-II scheduler function *OS_SchedNew()* was modified: 10 lines were added, of which 2 lines are compiler directives that allow the modification to be disabled or enabled from the OS configuration file. Finally, the *SchedulePolicy* variable (1 byte) was added to the TCB, in UCOS_II.H. An extra line was added to the TCB initialization function, *OS_TCBInit()* in order to set the scheduling policy to FPPS by default. The extra variable was added to the TCB for memory efficiency reasons. Alternatively, without adding code to the existing kernel modules, we could introduce a separate 'extended TCB' structure with a pointer that points back to the original TCB. The *ShouldSchedule* flag was added as a global variable, and is defined in OS_TEV.H, the header file which is shared by all the components of

our extension. An overview is given in Table 5 .

Name	Size (in bytes)	Where	File
ShouldSchedule	1	Global variable	OS_TEV.H
SchedulePolicy	1	Added to TCB	UCOS_II.H

LOC	Function(s)	File
1	*OS_TCBInit()*	OS_CORE.C
10	*OS_SchedNew()*	OS_CORE.C
80	FPDS Interface	OS_TEV_F.C

Table 5: Overview of memory footprint and lines of code for FPDS.

The code for FPDS support is OS-specific, as it depends on factors such as the implementation of the scheduler and the mechanism with which kernel functions are called.

5 RELTEQ

In this chapter we present the design and implementation of RELTEQ, a general timed event manager. In 5.1 we provide the motivation and requirements for our design. The architecture and design is described in 5.2, followed by the implementation in μC/OS-II in section 5.3. Finally we discuss the efficiency and modularity of in section 5.4.

5.1 Motivation

Considering the existing μC/OS-II kernel and our goals to extend it (with periodic tasks, and reservations), we identified the following timed events that need to be handled: delays, start of new period (job arrival), server replenishment, and server budget depletion. Since μC/OS-II only supports delay events, in order to handle all such events in μC/OS-II, we need a general mechanism for handling timed events. We set the following requirements for such a mechanism:

- Remain compatible with current μC/OS-II structures and architecture: (i) like other structures in μC/OS-II, set an upper bound on the number of structures (events in this case), and allocate them statically, (ii) if possible, extend existing structures rather than introducing new ones, while (iii) keeping changes in the kernel to a minimum.
- Small memory footprint. Use the same number of bits (16) to represent timestamps as μC/OS-II.
- Long event interarrival time. Prevent wrap-arounds from overflowing timestamps and the associated logic required to handle them.
- Only one hardware timer can be assumed, and should at least work with the periodic tick timer in μC/OS-II. Provide multiplexing of the events on a single hardware timer.
- Low overhead. Keep tick interrupt handling short.
- Prevent drift.
- Can handle various event types by defining an event type and associated event handler. We want to support delays, job arrivals, server replenishment and depletion. This should be extendable for future needs (e.g. deadline events for EDF scheduling).
- Can be extended to provide support for handling events in subsystems, which we look at in Chapter 7.

As explained in Chapter 2, the current timing features offered by μC/OS-II don't scale to multiple event types or cause too much overhead in the tick handler. Taking the same approach as [11] was also considered, but its absolute time model would not match μC/OS-II' use of relative timestamps. We also wanted to avoid having to deal with the extra logic for wrap-arounds this would introduce (see section 2.8).

Our proposal is the RELative Time Event Queue (RELTEQ), a general timed

event mangement mechanism, which supports long event interarrival time, no drift and low overhead. Based on a relative time model, RELTEQ is also able to replace the relative time-based timer features already provided in μC/OS-II. In this chapter we present the design and implementation, which we extend further in Chapter 7, to support HSF and reservations.

5.2 Design of RELTEQ

The main idea behind RELTEQ is to store timestamps in relative time. Unlike other relative time systems, however, where all event times are relative to the current time, event arrival times in RELTEQ are relative to each other. The arrival time of an event e_i is denoted as t_i. An event arrival time t_i is expressed relative to the previous event time t_{i-1}, with the first event e_0 being relative to some reference time, e.g. the current time. Like the TEQ extension to VxWorks [11], RELTEQ stores the events sorted by event arrival time in a linked list (queue). An example of a RELTEQ event queue is shown in 16. By storing the timestamps in such a relative way, we reduce the memory footprint required to store event timestamps. We also avoid wrap-arounds and their associated logic, reducing overhead.

As described in section 2.7.1, the drawback of using relative times is the need to update the timestamps as the current time increases. In the standard μC/OS-II kernel, this is done by the tick handler, which - on every tick interrupt - decreases all active relative timers by one tick. In the RELTEQ model, this problem is avoided as updating all events is not required. As only e_0 is relative to the current time, it is the only value that needs to be updated. The absolute time of an event e_i can thus be expressed by $t_{ref} + \sum_{k=0}^{i} t_k$, where t_{ref} is the current time (or any other reference time).

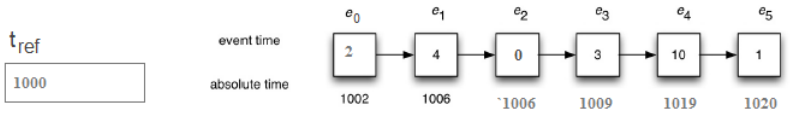


Figure 16: An event queue of relative timed events

The general architecture for RELTEQ is shown in Figure 17. It provides an interface to applications and other OS components for timed event management, requires a hook for the timer tick handler, and access to the OS API to implement the event handlers. As the implementation of the event handlers depend on the API of the OS, this code is OS specific, see section 5.2.4. We describe the different parts in the sections below. Implementation specifics are discussed in 5.3.

5.2.1 Events

The events module contains the event data structure and a table of events. The event structure contains the *timestamp* of an event, and a *pointer,* which may point to an object that is associated to this event (e.g. a task or server

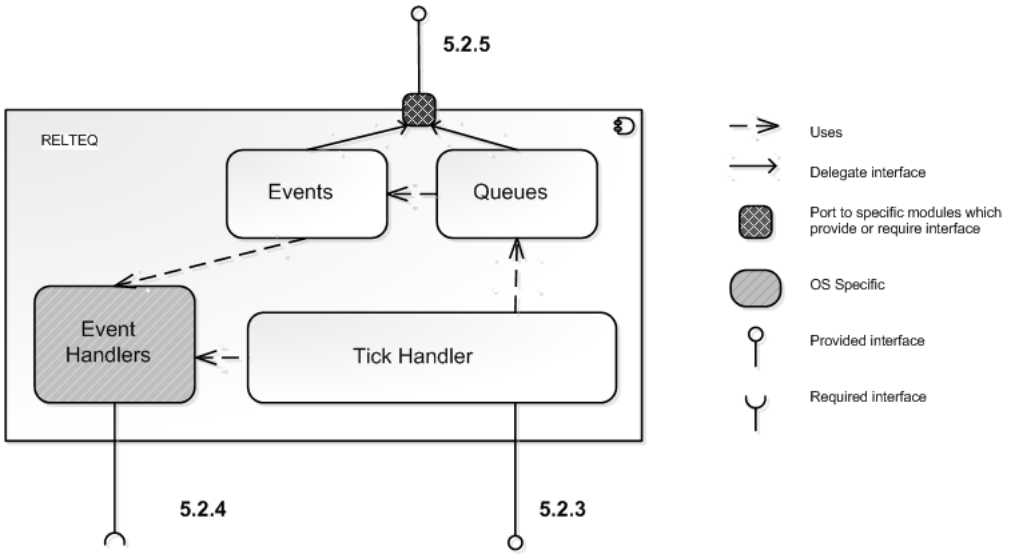


Figure 17: Component diagram of the RELTEQ architecture, consisting of queues, events, and multiplexing handling. The numbers indicate the sections describing the interfaces.

instance). Each event also has a corresponding *event type*, e.g. *periodic task arrival* or *delay*. One event instance can be created per event type per associated object (e.g. one periodic arrival event per task).

The size of the timestamp variable determines the maximum time period that can be expressed relative to the previous event. When using n bits to store the time, the maximum interval between two consecutive events is $2^n - 1$. This already improves on the circular model mentioned in Chapter 2, where the maximum interval between *any* two events is $2^{n-1} - 1$, meaning no event further than $2^{n-1} - 1$ time units away can ever be stored. In the worst case scenario, however, a large relative timestamp needs to be represented while not enough preceding events (to which this timestamp can be relative) are available. By inserting special *dummy events*, RELTEQ allows for an abitrary interval between any two events. When an event e_i and e_{i+1} are more than $2^n - 1$ apart, one or more dummy events ed_i, with times $td_o, td_1, .., td_m$ can be inserted between e_i and e_{i+1}. We can insert as many dummy events as needed, i.e. until the remaining relative time between the last dummy event $td_m$and e_{i+1} is smaller or equal to $2^n - 1$.

5.2.2 Queues

For handling queues of events we define a queue structure and several queue operations. Similarly to [11], we introduce a separate queue per event type. This reduces the average length of a queue, therefore increasing the insertion speed. This also makes it easy to iterate over all events of a certain type, e.g. for a deadline-based scheduler to access a sorted queue of deadlines. In Chapter 7 we also introduce per-server event queues. Note that *dummy events* are not considered a separate event type by itself, and can exist in every queue.

The queue structure itself defines a linked list of events of a certain event type. As multiple event queues can be created, a list of event queues, called *ActiveEventQueues*, is also stored, by adding a pointer to the next queue in the queue structure. We define the following operations on a queue.

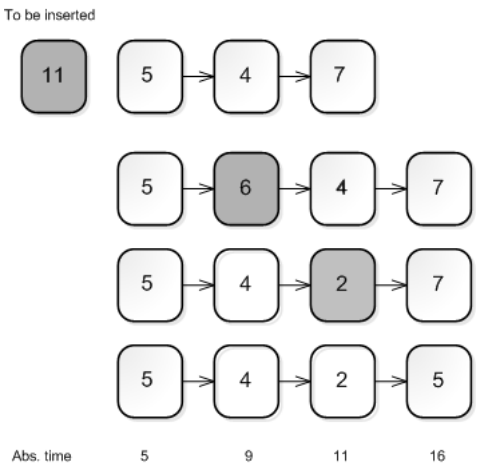


Figure 18: An example of inserting a new event with time 11, in a timed event queue with events at times 5, 9, and 16. The absolute times are shown for the last step of the process. Only forward links are shown (i.e. previous pointers are left out).

Insert. To insert a new event e_i at time t relative to the current time, the event queue is traversed, decreasing t with the relative timestamps of the traversed events, until a later event e_j is found for which its timestamp t_j is larger than or equal to the remaining value of t. e_i is then inserted before e_j, t_i is set to the remaining value of t, and t_j is decreased by t_i. When no such later event e_j exists, the event is inserted at the end of the queue. If the end of the queue is reached, and t is larger than $2^n - 1$, dummy events are inserted until $t \leq 2^n - 1$. Figure 18 shows an example of this process. Once the timestamps of e_i and e_j are updated, the linked list pointers are updated and e_i is now part of the queue.

Delete. An event is identified by its event type and associated object, e.g. *delay event* for *task 1*. Once the queue with the matching event type is found, we can traverse it to find the event e_i with the corresponding associated object. When e_i is removed from the queue, we have to update the timestamp of the next event in the queue, e_j. As e_j is stored to be relative to e_i, and we're removing e_i, e_j needs to become relative to the event e_i is currently relative to. This is done by adding e_i's timestamp, t_i, to e_j's timestamp, t_j. If $t_i + t_j > 2^n - 1$, an additional dummy event is inserted. This process is shown in Figure 19.

Pop first event from queue. To pop the first event e_0 from the queue, we simply remove it from the queue by updating the linked list pointer. Unlike a normal delete operation, the timestamp of the next event in the queue is not updated.

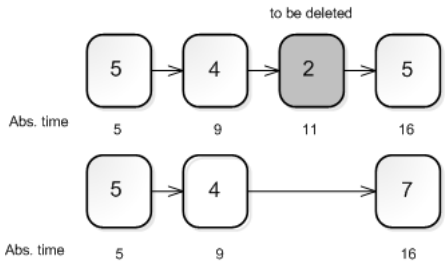


Figure 19: Example of deleting an event from an event queue

5.2.3 Tick Handler

The *tick handler* is responsible for checking for the expiration of any events, call their event handlers and update the queues. RELTEQ is not bound to a specific type of hardware timer, but requires a hook to invoke the tick handler on a timer interrupt. The temporal granularity and the timing precision depends on the resolution of the available hardware timer, and event times are thus expressed in ticks.

In section 2.7 we introduced one-shot and periodic timers. In the case of periodic timers, a timer interrupt is generated at every system tick, i.e. the frequency of the periodic timer. When a one-shot timer is used, a timer interrupt occurs only when the timer is programmed to expire. In the case of one-shot timers, we define the tick as the resolution of the hardware timer. As the tick handler does not fire every tick when using one-shot timers, the RELTEQ tick handler is a *dynamic tick* handler. We denote the number of ticks between the last invocation of the tick handler as t_t. When using a periodic hardware timer, like in μC/OS-II, $t_t = 1$.

The system contains n_q event queues. Let $e_{i,j}$ be the ith event in queue j. In this case we would have n_q events being directly relative to the current time, i.e. $e_{0,0}, ..e_{0,n_q-1}$. When using a periodic timer, to avoid having to update n_q relative times on every tick, we multiplex the queues by introducing a pointer p which points to the first event in a queue having the lowest relative time. The event to which p points we denote as e_p, with corresponding timestamp t_p. On every tick, we will then increase a multiplex counter t_m by t_t, until it is equal to the relative time value as indicated by t_p. When using a one-shot timer, the next invocation of the tick handler is always set to occur at t_p, such that $t_t = t_p$.

When t_p time units have passed (and now $t_m = t_p$), we decrease the queue heads accordingly (-t_p). For every queue for which the first event has timestamp 0, all events are popped from the queue and handled, until an event with a timestamp larger than 0 is found. The counter t_m is reset to 0, and p is set again to the head event of the queue with the lowest relative value, see Figure 20. All queue heads are now directly relative to the current time again.

Note that when a new event is inserted at or removed from the head of an event queue, pointer p is updated accordingly by calling a procedure *Multiplex()*.

The functionality of the tick handler is described by the pseudo code in figure 21. The interface between the OS and the RELTEQ tick handler is defined by:

TickHandler(t_t). Procedure to update all event queues and handle due events, where t_t is the number of ticks since the last tick handler invocation. Returns a tuple (t_p, *handled*), where t_p is the number of ticks to the next first due event, and handled is the number of handled events. When using a one-shot counter, it should be programmed to fire in t_p ticks.

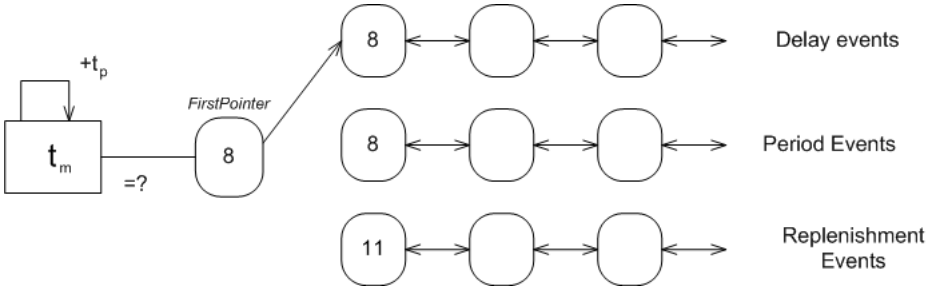


Figure 20: Multiplexing three queues of events, with $t_t = 1$.

TickHandler(t_t):
if $t_p \neq \infty$:
 $t_m = t_m + t_t$
 if $t_m = t_p$:
  DecreaseQueueHeads(N)
  t_m, t_p, *handled* = 0,∞, 0
  for q **in** *ActiveEventQueues*:
   while *Head*(q) **and** *Head*(q).*EventTime*= 0:
    e = *QueuePop*(q); *HandleEvent*(e); *handled* = *handled* + 1
   end while
   if *Head*(q): t_p =*min*(*Head*(q).*EventTime*, t_p)
  end for
  return (t_p, *handled*)
 end if
 return ($t_p - t_m$, 0)
end if
return (∞,0)

Figure 21: Pseudo code for *TickHandler(t_t)*. *Head(q)* returns the first event in queue q. *ActiveEventQueues* the list of queues, e.g. [*DelayEventQueue*, *PeriodEventQueue*].

5.2.4 Event Handling

An event is handled by its associated event handler, where each event type has a different event handler. Depending on the event type, different OS functions are needed to handle it, e.g. inserting or removing tasks to and from the ready queue, changing a status flag or releasing a semaphore. The handlers themselves are therefore OS specific, and require an interface to the OS' API.

5.2.5 Interface

Applications and other OS components can create and remove events through the provided interface.

GetEventQueue(et ***)***. Function which returns a pointer to the event queue for event type et.

QueueInsert(p, q, t ***)***. Procedure to create (allocate) a new event and insert it into queue q. p is a pointer to an object associated to this event (e.g .a task or a server). The event is due t time units from now. The type of the event depends on the type of the queue in which it is inserted. As an event is identified by its type and pointer p, if the queue already contains an event for p, it is replaced. In other words, it is only possible to add one event for each object (like a task or server) per event type. Although event timestamps are stored using n bits, t is allowed to be larger than $2^n - 1$ (dummy events will be inserted accordingly). As due events are handled on the next tick, t must be larger than 0 (events that are due now should be handled immediately).

QueueRemove(p, q ***)***. Procedure to remove the event for object p from queue q, if it exists. Once the event is removed from the queue, it is deleted (deallocated).

QueuePop(q ***)***. Procedure to remove the head event e from the queue q. Returns a pointer to e.

EventDelete(e ***)***. Procedure to delete (deallocate) an event e. To be used after an event has been popped from the queue with *QueuePop()*.

Demultiplex(). Procedure to make all queues relative to the current time again, by substracting the multiplexing counter t_m (see section 5.2.3) from the head of the active event queues. t_m is reset to 0.

Multiplex(). Procedure to scan the heads of the active event queues to find the event with the lowest event time, i.e. that expires first, and adjust the pointer p (see section 5.2.3) accordingly. When using a one-shot timer and the new value of t_p is different from the previous one, the timer should be reprogrammed to fire at t_p.

5.3 μC/OS-II Implementation

To extend μC/OS-II with RELTEQ, we added a separate OS module, OS_TEV_Q.C. Figure 22 shows an overview of the implementation of RELTEQ in μC/OS-II. The original μC/OS-II tick handler has been modified to call RELTEQ's tick handler instead. Some changes were made to the existing support for delays, which depends on the old tick handler (details are provided in section 5.4.3). We also extended the existing delay function, to support longer delays. Support for the new module can be switched on by enabling the OS_TEV_EN directive in UCOS_II.H.

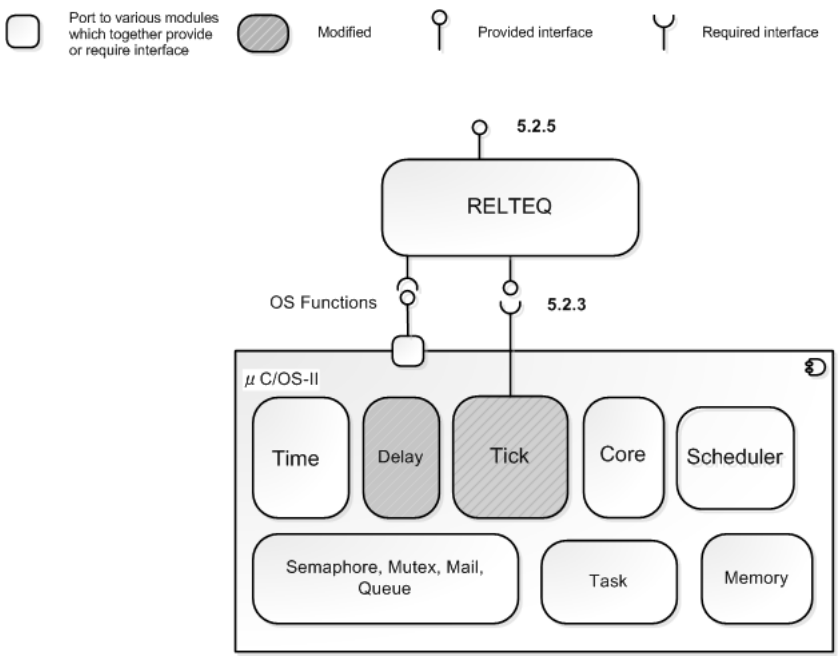


Figure 22: Overview of the RELTEQ implementation in μC/OS-II. The numbers indicate the sections describing the interfaces.

5.3.1 Data Structures

All the events are stored in the EventTable, which is a statically allocated array, similar to other μC/OS-II structures like the TCB table. Index pointers to events are simply numbers that specify the index in the EventTable where the event can be found.

For each timed event, several attributes need to be stored. In [11], a node in the TEQ contains the following information: task associated with the event, (absolute) timestamp, wrap-around flag, and two (previous and next) pointers for the double-linked list. For each queue of events, [11] stores the length, pointers to the first, last and middle event node, and the wrap-around status. To reduce the memory footprint for each event, our RELTEQ implementation will only store a 16-bit relative timestamp, an index pointer to the previous and next event (EventNext and EventPrev), and a pointer to the object associated to the event (server, task, or *none* in case of a dummy event). The type of an event node depends on the queue to which it belongs, and is not stored explicitly in every event node.

At initialization, the EventNext pointer of each event is set to point to the event next in the table. In order to keep track of the free (unused) events, an index pointer FreeList is introduced, which points to the first free event (initially: event at index 0 in the table). Whenever a new event is inserted into an event queue, the event currently pointed to by FreeList is used. FreeList is then updated to point to that event's EventNext pointer. When an event is deleted (after using *QueueRemove* or *EventDelete*), the event is returned to the list of free events. This is done by setting its EventNext pointer to FreeList,

and update FREELIST to point to the event. When an event is removed from the free list and inserted into a queue, its EVENTNEXT and EVENTPREV are used to point to the next and previous event in the queue. When the event is free, EVENTNEXT is an index pointer to the next free event, and EVENTPREV is unused.

The queue structure contains an index pointer to the first event in the queue, and an 8-bit value indicating the type of the events in this queue. It also contains a pointer to the next queue in the list of active queues. All queues are stored in a statically allocated array of queues.

The values defined for the multiplexer are FIRSTEVENT and TIMER, which correspond to the *FirstPointer* and t_m in Figure 20, respectively. Another boolean HASITEMS is defined which indicates if at least one event exists in any of the active queues.

In the first implementation of our extension, no support for reservations was built in yet. This meant that an event could only be associated with a task. In this case, a pointer to the object associated to the event can be left out by storing the timed events inside the TCB already available in μC/OS-II. The TCB will thus contain n_q previous and next event pointers, to form n_q queues. As the events are stored in the TCB, no additional memory structures have to be allocated. Although using less memory, it makes the solution less flexibile, and specific to μC/OS-II only; our implementation no longer follows this approach.

5.3.2 Tick Interrupt

As our extension should remain compatible with μC/OS-II, we use the periodic tick timer in μC/OS-II as the hardware timer. As μC/OS-II needs to remain portable to many platforms, this is the timer mechanism that is used on various ports. Next to μC/OS-II, which depends on the periodic timer to keep track of the system time, 3rd party applications built on top of μC/OS-II might also assume a periodic tick handler. To prevent drift (see 2.7.6), the event times are expressed in system ticks, and use the same grannularity as the periodic timer.

If a HPET timer (see 2.7.4) would have been available, this too, might have been used for as the hardware timer. As this timer can be programmed by setting an expiration time relative to the previous event, the same RELTEQ mechanism can be used as when using a periodic tick. We simply have to program the HPET timer to fire at the timestamp of the first upcoming event t_p.

In order to use μC/OS-II' tick interrupt as the hardware timer for RELTEQ, a hook is needed to call the tick handler from the timer interrupt. μC/OS-II provides one such hook, OSTIMETICKHOOK. As this is the only hook, we decided not to use it, as we should assume that the system programmer might depend on it for its own application or kernel extension. Another issue is that we want to skip the existing part of the tick handler where μC/OS-II updates the relative task counters. As RELTEQ will provide for timers anyway, executing this old part of the tick would be inefficient. We therefore decided to replace the tick handler by adding a jump to our own handler, after which the interrupt returns. Like the original tick handler, the new implementation also takes care of updating the system time and implements the mentioned hook for other extensions.

Instead of looping through all tasks and updating any active delay timers,

we perform the process explained in the Tick Handler section. When no events are due, the tick handler simply returns. This effectively reduces the clock tick routine to a virtual one-shot timer, but leaves the tick mechanism itself intact to be used for keeping track of time and allowing other μC/OS-II applications depending on the mechanism to use it.

When returning from the timer interrupt, *OSIntExit()* is called (see Figure 7). As the ready state of tasks and reservations can only be changed when an event has actually been handled, scheduling is not needed when no events were due. In this case calling the scheduler is skipped, and the currently running task is resumed immediately. The behavior of the timer interrupt is described with pseudo-code in Figure 23.

TimerInterrupt():
μC/OS-II Specific:
UpdateSystemTime()
OSTimeTickHook()
RELTEQ:
$t_p, handled = TickHandler(1)$
if $handled > 0$:
  Schedule()
end if

Figure 23: Timer interrupt pseudo-code.

5.4 Evaluation

In the previous section we have demonstrated how RELTEQ provides a mechanism to handle different types of timed events, with support for long event inter-arrival, and multiplexed on a single hardware timer. In this section we show that our design has low overhead, a small memory footprint, and is portable to other operating systems.

5.4.1 Processor Overhead

To show that RELTEQ causes little overhead, we run various test task sets. We also make a comparison between using RELTEQ's timed events to handle delays, or using the existing delay mechanism in μC/OS-II. Although it was not our main goal to replace the existing delay timer in μC/OS-II, by comparing the two timer mechanisms, we show that using RELTEQ instead causes little overhead, and in most cases even shows an overhead reduction. As μC/OS-II timer mechanism can only be used for delay events, the number of events in the test task sets is limited to one per task.

Total overhead equations:
The time spent on inserting and handling events[1] during the response time R (in ticks) of a task, with ε the number of due events during R, can be expressed by the formula:

[1]the general overhead caused by the timer interrupt is left out, as it is equal for both approaches, caused by by μC/OS' periodic timer interrupt. As RELTEQ does not necessarily depend on periodic timers, the overhead can be reduced further if a one-shot timer would be available.

$$\varepsilon * C_{handle_q} + \varepsilon * C_{insert_q} \quad (1)$$

where C_{handle_q} and C_{insert_q} are the worst-case costs handling and inserting an event with RELTEQ. Using the delay timer mechanism in μC/OS-II, this is:

$$\varepsilon * C_{handle_\mu} + \varepsilon * C_{insert_\mu} + R * N * C_{update} \quad (2)$$

where N is the number of created tasks in the system. C_{handle_μ} and C_{insert_μ} are the costs with μC/OS-II to handle and insert an event. C_{update} is the cost to update the timer counters in the tick handler.

Complexity of individual terms:
In RELTEQ, as events are stored in a sorted linked list of events per type, and there is at most one event instance per task per event type, the insertion time, C_{insert_q}, is $O(N)$, where N is the number of tasks. To pop an event from the list is $O(1)$, as only a linked list pointer needs to be updated.

For setting and updating delays in μC/OS-II, the insertion and update time, C_{insert_μ} and C_{update}, is $O(1)$ as it simply assigns a timestamp value in the task's TCB. Because of the additional queue management, both handling and inserting an event is expected to take more time in RELTEQ.

Tick handler overhead:
Using RELTEQ, we remove the need to update timestamps in the tick interrupt. To check for a due event, RELTEQ's tick handler only needs to check the multiplexed timestamp value. This moves most of the work of managing events out of the tick handler. As μC/OS-II' tick handler needs to update all timer counters, it is proportional to the number of tasks, $O(N)$, with N the number of tasks. This is reflected by the $R * N * C_{update}$ component in formula (2). As RELTEQ's tick handler only handles due events, it is proportional to the number of due events ε, i.e. $O(\varepsilon)$ in time. In the worst case, we gain nothing if all events need to be handled upon every tick. However, we assume that the supported events such as delays, periods, deadlines & replenishment are mostly used in intervals longer than a few ticks, thus we reduce the overhead compared to the other method.

Trade-off:
To express the trade-off, we show in which cases the overhead of RELTEQ is less than that of μC/OS-II, by comparing formula (1) to formula (2), i.e. when (1) $<$ (2). As $C_{handle_q} > C_{handle_\mu}$, and $C_{insert_q} > C_{insert_\mu}$, we substract $\varepsilon * (C_{handle_\mu} + C_{insert_\mu})$ from both sides, and get the following inequality:

$$\varepsilon * (C_{handle_q} - C_{handle_\mu} + C_{insert_q} - C_{insert_\mu}) < R * N * C_{update} \quad (3)$$

In the worst-case, every task has an active event, and every event that is handled immediately re-inserts itself. This means N events are active in the system at any time, and expire every δ ticks (the interval of the events), after which they are re-inserted. ε, the number of due events during T, can thus be written as $R * N/\delta$. Inserting this in the above inequality, we get:

$$\frac{C_{handle_q} - C_{handle_\mu} + C_{insert_q} - C_{insert_\mu}}{\delta} < C_{update} \quad (4)$$

Event interval $\delta = 1$ represents the absolute worst-case, in which every event is due and re-inserted on every tick. When the interval δ of events is large enough, RELTEQ's worst case will always be better than that of μC/OS-II's mechanism, for any number of events. This corresponds to our measurements with the test task set below.

Test task set:
In the following test we measured the response time of one task τ_c which performs some calculation (filling up a buffer with multiplications). The computation time C_c is 1 second. The task is preempted by a variable number of higher priority "sleep" tasks τ_{s_1} to τ_{s_n}, which continuously perform a sleep for a variable interval δ. As the τ_s tasks themselves don't perform any calculation, any difference in response time T_c for task τ_c can be attributed to the handling of the delays (including context switching). Pseudo code for the taskset is shown in Figure 24. We compared the difference in response times between using μC/OS-II and RELTEQ for handling the delays, see Figure 25. The response time was found by adding a profile point before and after the calculation in τ_c. The tick interrupt frequency is set to 100Hz, i.e. 1 second is 100 ticks. The number of events is equal to the number of sleep tasks, as every task creates one event: the delay event. When δ is really small, most of the cpu time is spent on handling delays and context switching. Especially with 40 tasks, the overhead becomes large as little time remains during every tick to perform actual calculation. As the interval δ gets larger, the speed increase of RELTEQ becomes larger as for most of the ticks no action is required, whereas the original tick handler needs to update all counters. When the delays become even larger, the total overhead from the events becomes very small, and not much more is gained with increasing intervals.

$\tau_{s_i}(\delta)$:
while *True:*
  Delay(δ)

$\tau_c()$:
profilePoint('before')
for i **in** [1,*IterationsPerSecond*]:
  buffer[i % *BufferSize*] = $(i * i)$ % i
profilePoint('after')
Stop()

Main(N, δ):
for i in [1,N]:
  CreateTask($\tau_{s_i}(\delta)$)
CreateTask(τ_c)
Start()

Figure 24: Pseudo code task set with two parameters: number of tasks N, and event interval δ. *IterationsPerSecond* and *BufferSize* are constants. *CreateTask* adds a task to the taskset, *Start*() starts the taskset, and *Stop*() terminates the application.

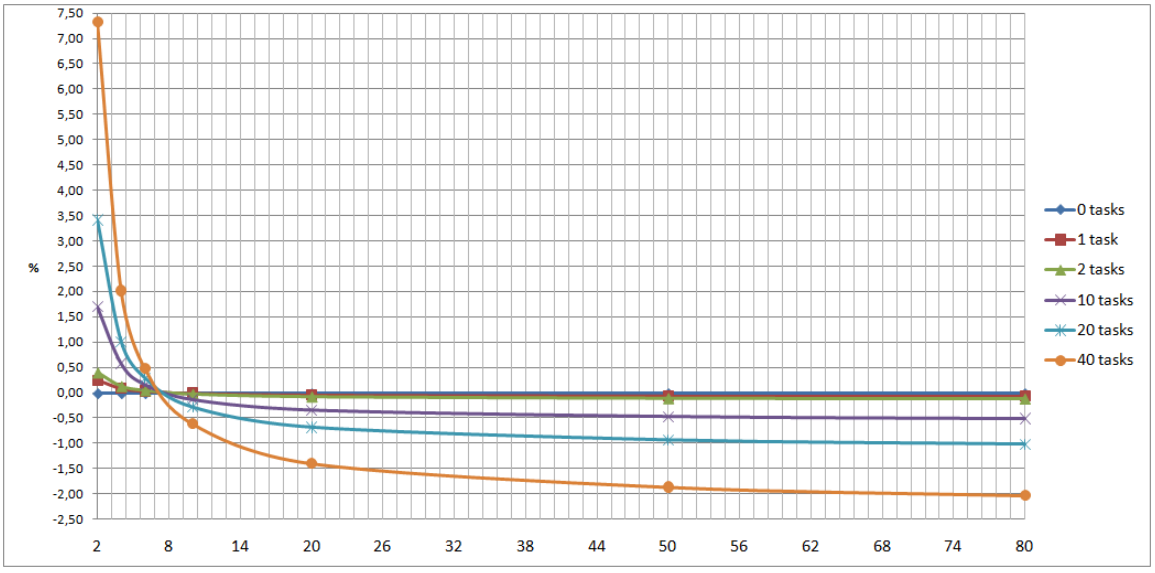


Figure 25: Overhead increase in % of RELTEQ vs μC/OS-II (negative: RELTEQ causes less overhead). Delay event δ interval in ticks on the horizontal axis.

In another test, we look at the computation time of the tick interrupt itself. Again we have a variable number of tasks which continuously perform a delay. We vary the number of tasks, and measure how many clock cycles are taken by the tick handler itself. As noted above, in Figure 26 we see that handling an

event is more expensive with RELTEQ due to the additional queue management. In Figure 27 we see that when no events are due, μC/OS-II' tick handler takes more time, as its handler is $O(N)$, rather than $O(e)$. This was measured by running the same task with $\delta = 10$, and measuring the tick interrupt at time $t = 5$, where no events are due.

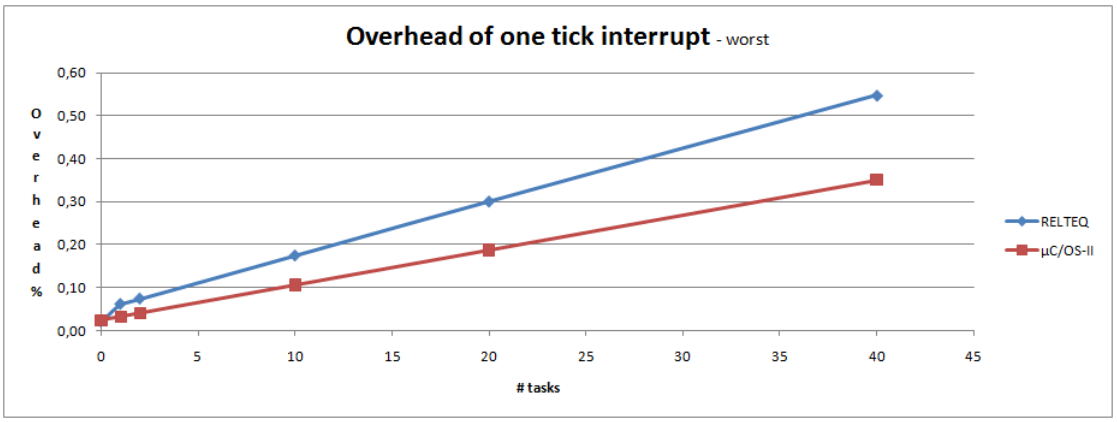


Figure 26: Overhead of the tick handler in one tick (in %). Worst case, number of due events is equal to #tasks.

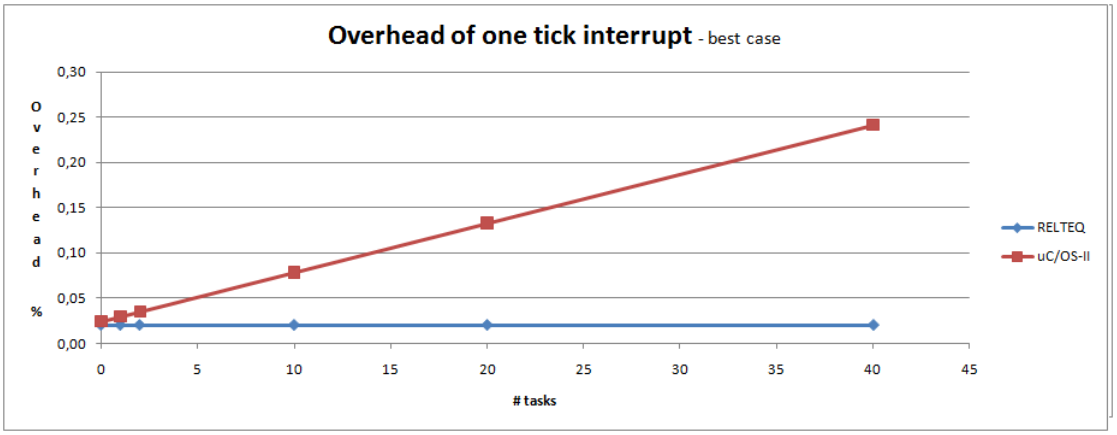


Figure 27: Overhead of the tick handler in one tick (in %). Number of due events is 0. Total number of events is equal to #tasks.

As shown in Figure 25, the additional overhead introduced by RELTEQ is low, and in almost all cases faster than μC/OS-II' counting mechanism for delay events. The worst cases in which the overhead is higher - when many events are periodically added with a very small inter-arrival time (t $<$ 10 ticks) - are unlikely to be very important considering the nature of these events. The total overhead of the tick handler is low as action is only required to handle due events; no timestamps need to be updated on every tick. It could be further reduced when another hardware timer is available, which would remove the need

of periodic tick interrupts altogether using RELTEQ's mechanism.

5.4.2 Memory footprint

As shown by tables 6.a and 6.b, each event (as defined in section 5.3.1) takes up 10 bytes. Every queue requires 7 bytes.

By using index pointers instead of regular pointers (like e.g. in [11]), we can save memory on most platforms like OpenRISC 1000, where a pointer takes up 32-bits (4 bytes), and an index pointer 16-bits (2 bytes). Depending on the hardware platform and the compiler, additional padding space might be added to the structures for memory alignment, which we do not include in the evaluation sections in this thesis.

Field	Memory (bytes)
EventNext	2
EventPrev	2
EventTime	2
ObjectPointer	4
Total	10

(a)

Field	Memory (bytes)
EventFirst	2
EventType	1
QueueNext	4
Total	7

(b)

Table 6: Memory footprint of (a) event structure, (b) queue structure.

5.4.3 Modularity

The design of RELTEQ is modular, so it can be enabled or disabled with a single compiler directive in the OS configuration file. RELTEQ, stored in a separate module, OS_TEV_Q.C, consists of 454 lines of code. The header file, OS_TEV.H is another 58 lines, and includes the event and queue structure definitions.

No original kernel code was deleted or modified. Instead, we inserted all our additions to existing kernel modules within compiler directive blocks, so they can be disabled easily. In order to use the tick handler as the hardware timer for RELTEQ, 4 lines were added to provide a hook in OS_CORE.C to call our custom tick handler. To use timed events for μC/OS-II' delay functions, we added 10 lines to the delay functions in OS_TIME.C. Another 3 lines were added to the initialization function, *OSInit*(), in OS_CORE.C to call RELTEQ's initialization function.

In order to port RELTEQ to another Operating System, custom event handlers need to be implemented (as marked as OS specific in Figure 17). Out of 454 lines, 11 lines of code are currently needed for the implementation of a μC/OS-II delay handler. Furthermore, a hook would have to be provided in the tick handler, to call the RELTEQ tick handler instead. Our code also assumes a function to disable and enable interrupts; this would have to be replaced with the equivalent function for the target platform.

6 Periodic Tasks

As mentioned in section 2.2, a periodic task τ_i is described by a tuple (ϕ_i, T_i, C_i, D_i). We will assume the deadline is equal to the period ($D = T$). After the initial offset time ϕ_i, a new job instance is released every T_i time units.

In order to add support for the periodic task model to μC/OS-II, we need to introduce primitives to (i) register a periodic task with its parameters ϕ_i and T_i (C_i is implicit), (ii) wait for the next job instance, (iii) inform a task about deadline misses. To this end, we want to provide the interface described in the next section.

6.1 Interface

The interface to manage periodic tasks consists of the two procedures described below. Initially, after a task has been created, its period, T_i is initialized to 0, indicating the task is not periodic.

TaskSetPeriodic(τ_i, T, ϕ). Procedure to register the task τ_i as a periodic task with offset ϕ (relative to the time this procedure is called), and period T. After the task is registered, the first job is released at ϕ, and every subsequent job arrives every T time units. Pre-condition: a task exists at priority i, $T > 0$, and$\phi \geq 0$. Post-condition: $T_i = T$, and $\phi_i = \phi$.

WaitForNextPeriod(). Procedure to wait for the next task instance. When a deadline has been missed, it returns immediately. The procedure returns the number of missed deadlines. Pre-condition: $T_{cur} > 0$.

6.2 Design & Implementation

To implement periodic tasks, we will discuss various alternatives below. In each case, k is the instance number of the job, and the actual work of the job is represented by the procedure $f_i(k)$. As shown in Chapter 4, a job can be split up further into subjobs. In this Chapter, we assume one subjob, which is implemented in $f_i(k)$.

In RTOSes like μC/OS-II, which have no built-in support for periodic tasks, the task is made periodic by manually sleeping until the release time of the next job. This case is illustrated by figure 28.

```
TASKτ_i():
k = 1
Delay(φ_i)
while True:
  f_i(k)
  k=k + 1
  Delay(T_i)
end while
```

Figure 28: Periodic task implementation (a), using delays.

An important downside of this approach is that it may give rise to drift in cases like μC/OS-II, where *Delay()* is implemented with a relative timer. The offset of the first task instance might also become much longer than ϕ_i, as the task first needs to be activated before the delay can start. We also need to manually compare timestamps to establish if we overran our deadline.

The authors of [5] also describe a potential problem with an implementation based on POSIX timer signals, see Figure 29. The POSIX timer signal is created with *timer_create()*, and programmed to send a signal every T_i time units with *timer_settime()*. The POSIX timer fires asynchronously, i.e. independent of the execution of the task τ_i. When the task body takes longer than T_i and signals arrive when the task is still executing, the signals might go unnoticed by the task (i.e. are lost). We have to make sure we detect all new period arrivals. Also note that when the task is preempted before *timer_settime* is run, the offset can be much longer than ϕ_i.

```
TASK τ_i():
k = 1
timer_create(CLOCK_REALTIME, NULL, timer_id)
timer_settime(timer_id, relative, φ_i, T_i)
while TRUE:
  sigwaitinfo(SIGALRM)
  f_i(k)
  k=k+1
end while
```

Figure 29: Periodic Task implementation (b), using POSIX signals

In order to prevent drift of periodic arrival times and keep track of missed deadlines, we extend the OS with the primitives mentioned in the previous section. *TaskSetPeriodic()* registers the task as a periodic task by setting the task's period variable, and using RELTEQ, it creates a timed event which is set to expire every T_i time, and initially at the phase of the task, ϕ_i. Each periodic task is assigned a period semaphore. On every expiration of the periodic timer, the periodic semaphore is increased, and a new periodic event is inserted to fire in T_i ticks. *WaitForNextPeriod()* blocks on this semaphore. As *WaitForNextPeriod()* marks the end of the current job instance, when it is called and the semaphore count is already higher than 0, we missed our deadline. Trying to lower the semaphore will thus succeed instantly, and a value is returned to indicate how many deadlines were missed. Note that when $\phi_i = 0$, the semaphore is initialized to 1.

As the pattern of looping and waiting for the next period (as shown in Fig. 30) is common for most periodic tasks, an additional abstraction could be provided where tasks are simply registered by calling *TaskSetPeriodic*($f_i()$, τ_i, ϕ_i, T_i).

Figure 31 shows the architecture of the periodic task implementation. The *Periodic Tasks* component contains the procedures to register a periodic task, and wait for the next period. The periodic event handler is added to the Event Handlers, as seen in Figure 17. For each task, we have to introduce two addi-

TASKτ_i():
$k = 1$
while TRUE:
 $missedDeadlines = WaitForNextPeriod()$
 $f_i(k)$
 $k{=}k + 1$
end while

MAIN():
TaskSetPeriodic(τ_i, ϕ_i, T_i)

Figure 30: Periodic Task implementation (c)

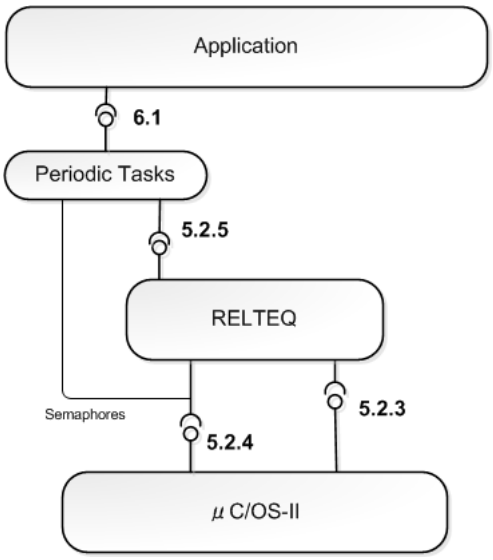


Figure 31: Architecture of periodic tasks in μC/OS-II using RELTEQ

tional variables: the period semaphore and an integer with the period length. For space efficiency reasons we added these variables to the existing TCB structure, but they could be declared in their own structure for modularity reasons.

6.3 Evaluation

6.3.1 Modularity & Memory footprint

The periodic task component is implemented in a separate file, OS_TEV_P.C. It consists of 94 lines of code. The periodic event handler is an additional 9 lines. As we add two additional variables to the TCB, 2 additional lines were added to OS_CORE.C (for initialization) and UCOS_II.H (TCB definition). The period length variable is 4 bytes. The size of semaphore variable depends on the configuration settings of μC/OS-II.

6.3.2 Behavior

To demonstrate the behavior of the implementation of the periodic tasks, we created a test task set with the parameters as shown in Table 7.

Priority	C	T	ϕ
1	15	25	5
2	17	50	-

Table 7: Parameters of periodic test task set. Values are in ticks.

As can be seen from the execution trace of the task set on the OpenRISC 1000 simulator in Figure 32, a job for task 1 arrives at times 5, 30, 55, and 80. Those for task 2 arrive at 0 and 50.

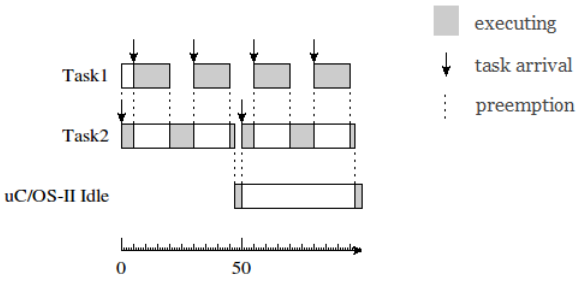


Figure 32: Execution trace of two periodic tasks

To demonstrate the correct handling of deadline overruns, we change the scheduling policy of the lower priority task. Task 2 will now be run using FPDS, so that it will finish its entire job before yielding to task 1. This causes task 1 to miss its deadline, see Figure 33 (a). *WaitForPeriod* correctly detects this and instantly returns. Another example is shown in (b), where a high load task set under FPDS is shown, in which task 1 misses multiple deadlines, but catches up as the system keeps track of the number of period arrivals.

6.3.3 Processor Overhead

To measure the overhead of handling periodic tasks, we added measurement points to the procedures *WaitForNextPeriod*(), *TaskSetPeriodic*(), and around the RELTEQ tick handler which handles the periodic events. The example task sets from the previous section, together with the added measurement points, were used to get the WCET (Worst Case Execution Times) of the different periodic task handling parts, as seen in table 8. The measured values are: C_{setup} for *TaskSetPeriodic*(), C_{event} for managing and handling the periodic events with RELTEQ, and C_{wait} for *WaitForNextPeriod*().

For every periodic task, we need to set it up once, and require one call to *OSTaskPeriodWait*() and the handling of one period event per period. The over-

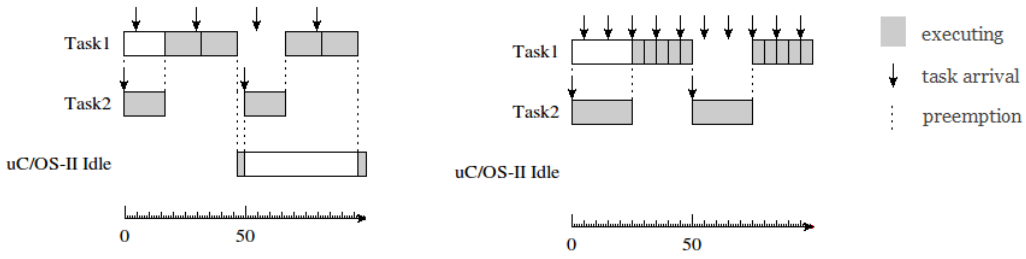


Figure 33: Deadline misses are detected correctly. (a) Run trace of same periodic tasks as in Figure 32, now in FPDS. (b) Another task set in which multiple deadlines are missed (also FPDS). Task parameters$\tau_i(T_i,\ C_i,\ \phi_i)$: τ_1(25,15,5) and τ_2(50,17,0).

	C_{setup}	C_{event}	C_{wait}
Cycles	5670	10704	4804
μs	56.7	107.0	48.0

Table 8: WCET of periodic task handling routines.

head for C_{event} is pessemistic, as RELTEQ takes less time to handle multiple periodic events in one batch (C_{event} includes the time to handle the event, perform multiplexing and queue management). To put the overhead into formula we get:

$$overhead(t) = N * C_{setup} + \sum_{i=1}^{N} \lceil t/T_i \rceil * (C_{event} + C_{wait})$$

N is the number of periodic tasks, and t the total running time of the application.

For the example set τ_1(25,15,5) and τ_2(50,17,0), which we ran for 100 ticks, we get $2 * 5670 + 100/25 * (10704 + 4804) + 100/50 * (10704 + 4804)$, which is 104388 cycles, equal to 1.04ms, or 0.1% of 100 ticks.

Figure 34 shows the worst case overhead of different periodic task sets with n number of tasks, in which each task has a period T. The total runtime for each taskset is 1000 ticks (10 seconds). The exact overhead numbers are shown for $n = 10$.

When using the mechanism described in Fig. 28 to create periodic tasks in μC/OS-II, the μC/OS-II delay timer mechanism would be used. As shown in section 5.4.1, using RELTEQ to handle (periodic arrival) events will result in less overhead in most cases.

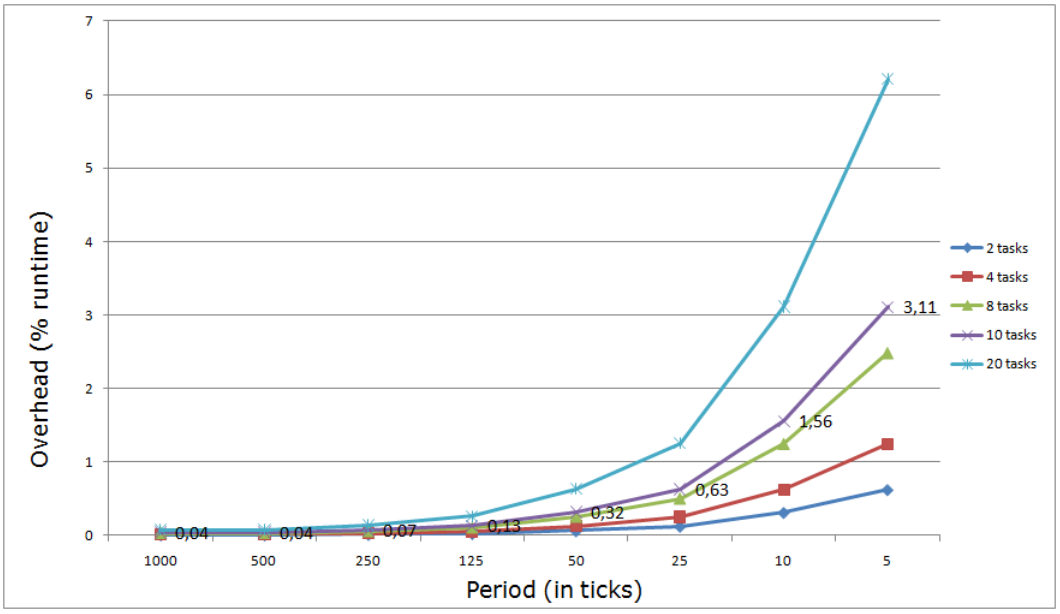


Figure 34: Worst case overhead of periodic tasks on OpenRISC 1000 simulator, at 100Mhz.

7 Reservations

In this chapter we will look at extending μC/OS-II with processor reservations based on fixed priority bandwidth preserving servers. We implement both the idling periodic and deferrable server. We also implement the polling server to demonstrate that our design is also suitable for non-bandwidth preserving servers. By creating a server σ_i, our design should allow the application to request a processor reserve for Θ_i processor time every Π_i time units. The interface which we want to provide to the application is described in section 7.1.

We make the following assumptions for our design:

A1. No more than 8 servers are needed, and 8 tasks per server are sufficient. Only one server can be assigned to each priority level.

A2. Tasks are not migrated between servers during run-time, and a task has exactly one parent server.

A3. Budgets are not renegotiable, and admission testing is done offline.

A4. Shared resources between servers can only be accessed by FPDS jobs (using FPDS as a guarding mechanism for critical sections, see section 7.6).

A5. Fixed priority scheduling is acceptable on both local and global level (but other scheduling policies might be added in the future).

A6. A subsystem cannot contain other subsystems.

A7. Only one subsystem is active at the time (single processor).

7.1 Reservation Interface

The application and other components can create servers through the following provided interface.

ServerCreate(*i*, Θ_i, Π_i, *type)*. Procedure to create a server σ_i, where i is the server's priority, Θ_i its capacity (in time units), and Π_i its replenishment period (in time units), with $1 \leq i \leq MaxServers$, and $0 < \Theta_i \leq \Pi_i$. *Type* specifies whether the server is idling periodic, deferrable or polling. Pre-condition: no server exists at priority i. Post-condition: a server σ_i is created at priority i, with $\sigma_i.\Theta = \Theta_i$, $\sigma_i.\Pi = \Pi_i$, $\sigma_i.type = type$.

ServerNameSet(*σ_i, name)*. Procedure to assign a name given by the text string *name* to a server σ_i. Pre-condition: a server exists at priority i. Post-condition: $\sigma_i.name = name$.

ServerMapTask(*σ_i, τ_j)*. Procedure to create a mapping between server σ_i and task τ_j, where i the server's priority, and j is the priority of the task in the server's taskset $\gamma(\sigma_i)$. Pre-condition: τ_j is currently not mapped to a server, $1 \leq j \leq MaxTasksPerServer$. Post-condition: τ_j is mapped to server σ_i.

As we use a static task-to-server mapping (which we describe in section 5.3), the assignment of tasks to servers is implicit: a task's priority level determines its

parent server. Therefore, our implementation only returns the global task priority k to be used directly with μC/OS-II's function to create a task, *OSTaskCreate(k)*. For example, *"ServerCreate(i,* Θ_i,Π_i*); OSTaskCreate(ServerMapTask(*σ_i, τ_j*))*;" creates a server σ_i, and a task τ_j in server σ_i.

7.2 Architecture

In order to provide support for processor reservations to applications running on top of μC/OS-II, we identified (see section 2.5.2) two mechanisms which we need to implement: servers (section 2.4) and hierarchical scheduling (2.6).

Support for several event types is needed to deal with the enforcement and accounting of server budgets. In Chapter 5 we looked at a general timed event mechanism, which we extend in this chapter to deal with events whose expiration times are relative to the consumed budget. A further extension is introduced to minimize the interference from events local to inactive servers and defer handling these events until the corresponding server is activated. Finally, to schedule the servers we introduce a two-level hierarchical scheduler. The μC/OS-II scheduler was modified to provide a hook through which our hierarchical scheduler is called, instead of the normal μC/OS-II scheduler. Figure 35 shows an overview of the architecture of processer reservations implemented through servers based on RELTEQ. Summarized, support for resource reservations consists of (i) an extension to RELTEQ (section 7.3), (ii) implementation of servers based on RELTEQ, (iii) an interface to create these servers (section 7.1), and (iv) hierarchical scheduling (section 7.5). Furthermore, we extend the FPDS component to be used with a hierarchical scheduler, which we denote as H-FPDS (section 7.6).

7.3 RELTEQ extensions

7.3.1 Queue (de-)activation

In Chapter 5 we introduced a list of event queues, with one event queue per event type. So far, two queues have been introduced: one for delay events and one for period arrival events. With the introduction of servers, we could add the delay and period events for all tasks in each server to the same two queues. However, this would cause the tick handler to handle events from any server as soon as they are due. As events are handled within the budget of the currently running server, if the due event concerns a task in an inactive server, it causes interference on the currently running task, i.e. the overhead for handling events in incactive servers is accounted to the currently active server. We therefore separate the events belonging to different servers into different queues. Certain queues contain event types that should always be handled, regardless of the currently active server. These are called *global event queues*, to distinguish them from *local event queues*, which contain events that should only be handled in the context of their own server. In the previous chapters, all queues were global, as servers did not exist.

We still keep a list of active queues, but allow the system to remove and add a list of active queues during run-time. As can be seen in section 5.3.2 the tick handler only handles events from the active queues. Whenever a server is

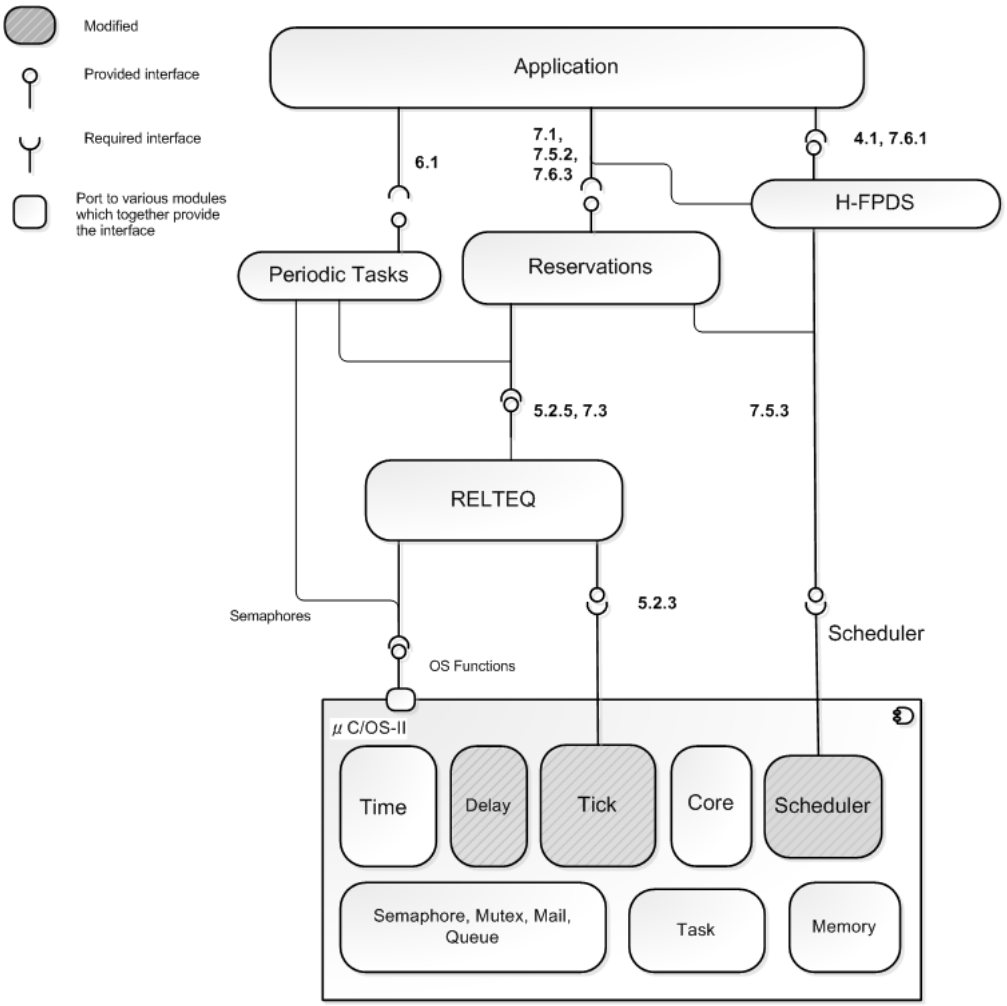


Figure 35: Architecture of Reservations (Ch. 7), periodic tasks (Ch. 6), RELTEQ (Ch. 5 and section 7.2) and FPDS (Ch. 4 and section 7.6) in μC/OS-II (Ch. 3). The numbers indicate the sections describing the interfaces.

switched in or out, its queues are added to (*activated*) or removed from (*deactivated*) the list, respectively. As a server can contain a list of queues that should be activated or deactivated, we introduce the following new procedure:

ActivateLocalEventQueues(ql**)**. Procedure to activate the specified list of queues, where ql is the head of a list of event queues. Post-condition: the queues in the queue list defined by ql are activated. Any list of queues previously activated by this procedure, is deactivated before activating ql. When called with $ql = \emptyset$, the procedure only deactivates the previously activated queues.

The active queue list always consists of the global queues, appended with the

activated local event queues. In fact, the only operation the procedure above has to perform is to update the next pointer of the last global queue to point it to the head of the activated queue list, ql, see Figure 36.

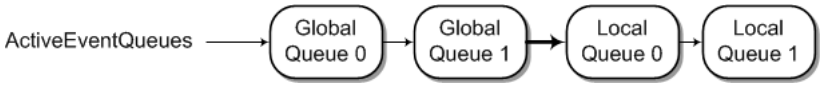


Figure 36: Active event queues, consiting of global queues, appended by a list of 0 or more local queues. The procedure *ActiveLocalEventQueues()* updates the pointer indicated by the thick arrow.

7.3.2 Queue synchronization

When an event queue is reactivated after it is has been deactivated for a period of time t_o, it is out of sync with the current time. The head of the queue (i.e. the first due event in the queue) is still relative to the time when the queue was deactivated, i.e. $-t_o$. As the local queues are deactivated whenever a server is switched out, when the server is switched back in and its local queues reactivated the head events in the local queues are relative to $-t_o$. This means that all events e_i with event times $t_i \leq t_o$ have expired. To keep track of t_o (i.e the time a server has not been running), we introduce the *stopwatch queue*, which we describe in the section, 7.3.3. To synchronize reactivated local queues we define the following procedure:

SyncEventQueues(σ_i, q_s). Procedure to synchronize the list of local queues ql of server σ_i. q_s is the stopwatch queue whose event times are accumulated to determine t_0, as further explained in the next section. Pre-condition: a server exists at priority i. Post-condition: For any queue in ql, all events that occured during time interval t_0 are popped and handled, and the head of the queue made relative to the current time.

Note that when an event e_i is handled while synchronizing the queue, this might cause another event e_j with timestamp $t_j \leq t_o$ to be inserted. These events are also handled until the entire queue is synchronized. For example, a periodic event which fires every 10 time units, was about to expire in 5 ticks at the time its queue was deactivated. 20 time units later, the queue is reactivated and synchronized. The periodic event is handled at time -15, and reinserts a new periodic event, to expire 10 ticks later. This event, too, is handled, at time -5. The next period event is thus due in 5 time units.

7.3.3 Stopwatch Queue

In order to keep track of time that has passed since a certain moment, we introduce *counter queues*. Unlike other event queues, upon every tick, the head of a counter queue is increased instead of decreased. By also using event queues to store counters, we can use the existing mechanisms to deal with storing large timestamps. We extend RELTEQ with the following methods:

IncreaseCounterQueue(q, $a = 1$). Procedure to increment the head event e_0 in queue q by a time units, where a defaults to 1 if no specific value is specified, and $0 < a < 2^n$. Post-condition: Timestamp t_0 of head event e_0 is increased by a. When t_0 would overflow (i.e. $t_0 \geq 2^n - a$), a new dummy event is inserted at the head of the queue, with timestamp a.

ResetCounterQueue(q). Procedure to reset the counter by removing all events. Although the same could be accomplished by using *QueueRemove()* on all events in the counter queue, this method is faster as all events are simply marked as free, and only the queue head pointer is updated, without updating timestamps and link pointers between events in the queue. Post-condition: all events from queue q are removed, except for the head event e_0, whose timestamp t_0 is reset to 0.

Counter queues in the active queue list are incremented by n by the tick handler, where n is the number of ticks passed since the last invocation of the tick handler.

In the previous section, we already identified the need for one such counter, the stopwatch, to keep track of the time t_o which has passed since the server was last switched out. During the time when a server is inactive, several other servers may also have switched in and out. Therefore, for each server, we need to keep track of how long it was inactive. Rather than storing a separate counter queue for each server, we exploit the RELTEQ approach of storing events relatively in a queue, and multiplex the stopwatches for all servers in one queue, the *stopwatch queue,* q_s. We do this by inserting a *stopwatch event* e_{s_i} in q_s, with time t is 0, whenever a server σ_i is switched out. The pointer e_p for the event points back to σ_i. When the server σ_i switches back in, we call the synchronization procedure *SyncEventQueues*(σ_i, q_s). Like for any other event queue, to find the timestamp of an event e_i, we accumulate the timestamps of all events e_j, with $j \leq i$. When we do this for the stopwatch event e_{s_i} for server σ_i, we find the time t_o, the time period for which the server has been inactive. After the synchronization method is called, the stopwatch event for server σ_i is removed from the stopwatch queue, using the *QueueRemove()* procedure.

Example of the stopwatch behavior The stopwatch queue is a good example of RELTEQ's versatility. It provides an efficient and concise mechanism for keeping track of the inactive time for each server. Figure 37 demonstrates the behavior of the stopwatch queue for an example system consisting of three servers A, B, and C. It illustrates the state of the stopwatch queue at different moments during execution. Whenever a server σ_i is switched out, and another server σ_j is switched in, we show the state of the stopwatch queue after σ_i is switched out but before σ_j is switched in (legend: *before switch in*), and after σ_j is switched in (legend*: after switch in*). Initially, when a server σ_i is created, a stopwatch event e_i is inserted into the stopwatch queue, with its pointer pointing to σ_i and a timestamp of 0. At time 0, when server A is switched in, the stopwatch event e_A for server A is removed from the queue. At time 3, when server A is switched out, its stopwatch event e_A is reinserted with time 0. Note that during the time when server A was running, the tick handler was incrementing the head of the stopwatch queue, which happened to be the stopwatch event for server B. At time 12, C is switched out, and its stopwatch event e_C is

reinserted with time 0. Then, server B is switched in, its stopwatch event e_B is removed from the queue, and the timestamp of e_A, which was relative to e_B, is adjusted accordingly. When server A is switched in at time 17, the time t_o it has been inactive is 14 ($e_{C_t} + e_{A_t}$).

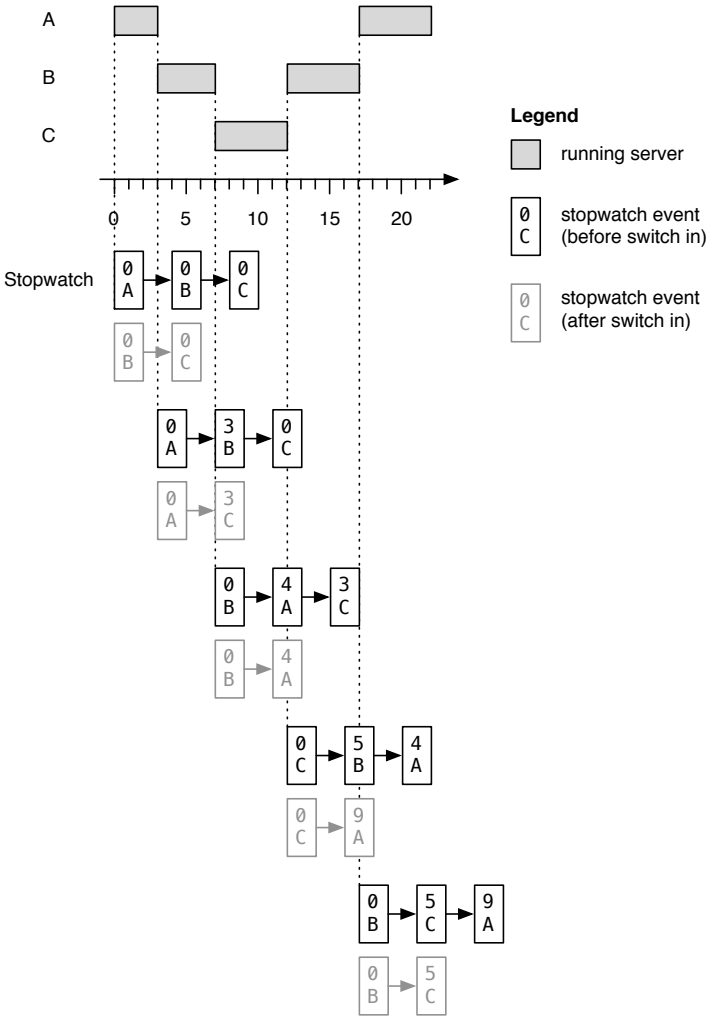


Figure 37: Stopwatch queue

7.3.4 Virtual Time

As each server is assigned a share of the processor time, the tasks running on the server can be seen as running on a virtual processor. Figure 38 shows an example of a server which is switched out at time $t = 6$, and switched back in at $t = 11$. Inside the server runs a task which fires an event after every 4 time units it has run. The *virtual times* (i.e. relative to the consumed budget) for this event are 4 and 8, while the wall clock times are 4 and 13. In other words, virtual events are triggered relative to the consumed budget. In the example,

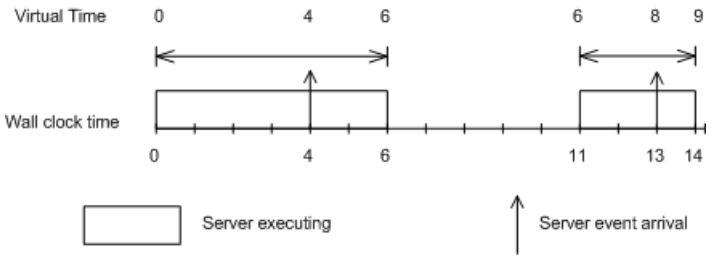


Figure 38: Virtual time

at time $t = 4$, a new event should expire 4 time units later, which is at time 8. However, in a traditional OS with events stored as absolute time, when the server is switched out, this event timer should be cancelled. Then, after the server is switched back in again, we find out the server has been inactive for 5 ticks, and have to update all the event timestamps (i.e. increase them by 5 ticks, so timestamp 8 becomes 13). We prevent this problems by exploiting the fact that only the head event of a RELTEQ queue is relative to a reference time.

We introduce an additional flag to each queue, IsVirtual. When set, it indicates this queue represents a virtual timed event queue. Similarly to queues introduced earlier, a virtual queue is deactivated when a server is switched out, and reactivated when the server is switched back in. The only difference is that a virtual queue is not syncronized with the stopwatch time, since the server does not consume any of its budget when it is inactive.

7.4 Servers

In this section we discuss the implementation of servers, based on the RELTEQ extensions introduced in the previous section 7.3.

7.4.1 Replenishment and Depletion

To support servers we create two new event types: *budget replenishment* (also called *server period*) and *budget depletion*. The queue for the budget replenishment events, is added to the global queues. When a server σ_i is created, a budget replenishment event is inserted with its pointer pointing to σ_i and its event time equal to the server's replenishment period $\prod_i$. When the event expires, the event handler reinserts the replenishment event at time $\prod_i$ and changes the servers state to *ready* if it was *depleted*, or leaves the state unchanged otherwise (e.g. *waiting*). The budget β_i is set to the server's capacity Θ_i.

Every server has a local virtual event queue for the depletion event. When a server is replenished, a new depletion event is inserted, with its pointer p pointing to σ_i and its time equal to the server's capacity Θ_i. If the server was not yet depleted, the old depletion event is removed by the *QueueInsert()* method. When the depletion event expires, the depletion event handler changes the server's state to *depleted*.

7.4.2 Server types

To limit the interference from inactive servers, when an active server is switched out (e.g. a higher priority server is resumed, or the active server gets depleted), its local server queues are deactivated. During the time the server is inactive, its events are not handled, and will therefore not interfere with the currently active server. When a server is switched back in, its local server queues are syncronized and activated, as explained in section 7.3.2. Depending on the server type we also have to deal with events of servers moving into the *waiting* state (see section 2.4). We describe the implementation of the different server types below.

Deferrable Server. When the workload of a deferrable server σ_i is exhausted, i.e. there are no ready tasks in $\gamma(\sigma_i)$, the server is switched out, its local server queues deactivated and the server is moved to the waiting state. Consequently, any periodic arrival event or expiring delay event that might wake up the server and put it back into the *ready* state cannot be noticed. One way of solving this problem is keeping the server's event queues active when it is switched out. This, however, would cause the length of the active queues list to be proportional to the number of deferred servers. As the multiplexer (see section 5.2.3), which is called by the tick handler and the scheduler, scans all these queues, its complexity would become linear in the number of deferrable servers too. Furthermore, the overhead of handling due events in the deferrable server queues would be counted against the currently active server. Instead, to limit the interference of inactive deferrable servers on the active server, we introduce *wakeup event*s, stored in a global event queue. When the workload of a deferrable server σ_i is exhausted, we deactive its local queues, change its state to *waiting*, and insert a wakeup event into the global wakeup event queue. The event's pointer e_p points to σ_i and its event time is equal to the arrival time of the first event in any of the non-virtual local event queues of σ_i. When the wakeup event expires, the state of σ_i is set to *ready*, and the handling of the corresponding local event is deferred until σ_i is switched back in.

Polling Server. As soon as a polling server σ_i switches out because its workload is exhausted, its remaining budget β_i is discarded (i.e. set to 0), and its state changed to *depleted*. As it can never reach the waiting state, no wake up events are needed. All the events that were expired but unhandled due to the server being switched out (e.g. due to preemption) are handled as soon as the server is switched back in.

After a server is replenished, its state is restored to *ready*. Unlike the other two servers, however, the polling server immediately goes into *depleted* state if it has no ready tasks. This means that after the server is replenished, switched back in, and synchronized, it might still have no ready tasks. Therefore, we perform an extra check when the server is switched back in: if, after synchronization, none of its tasks is ready, the server's state is set to *depleted*, and the scheduler should pick a new server (or the currently running one) to switch in (causing the polling server to be switched out again).

Idling periodic server. The idling periodic server is a polling server with an additional idle task (with the lowest priority). As there is always one task to run, it will never reach the waiting state, nor will it ever discard its remaining

budget. The idle task is added to the server upon creation, and idles away its budget when no other tasks in the server are ready.

One special idling periodic server is the *idle server*. This server is created when the system starts, at the lowest priority. It is switched in when no other servers are ready to run, and has infinite budget so it will always be ready.

7.4.3 Queues

At any time, the list of active queues consists of the global queues and the local queues of the active server, which are summarized in Figure 39. Globally, we introduced a stopwatch queue (see section 7.3.3), an event queue for wake up events (see section 7.4.2) and replenishments (see section 7.4.1). The local queues consist of the period arrival events, delay and depletion events.

7.4.4 Switching servers

Just like a context switch between the current task and a newly arrived highest priority ready task, we call switching out the current server and switching in another one, a *server context switch*. The functions for performing a server context switch, including switching a server in and out, are summarized by the pseudo code in Figure 40, 41, and 42. q_e represents a queue for event type e, e.g. q_s is the stopwatch event queue. For the definition of the queue functions, see section 5.2.5. Beforing switching out a server σ_i, *Demultiplex()* makes all currently active event queues (including σ_i's local event queues) relative to the current time. This is needed to ensure the queues will be relative to $-t_o$ when they are switched back in, rather than $-(t_o+t_m)$, with t_m the multiplex counter introduced in section 5.2.3. $\sigma_i.ql$ is used to denote the list of local event queues for server σ_i. *FirstNonVirtualEventTime(ql)* returns the timestamp of the event which is due first in any non-virtual event queue of a given queue list ql. Finally, σ_{hps} is the highest priority ready server, and σ_{cur} the currently active server.

Demultiplex()
ServerSwitchOut(σ_{cur})
ServerSwitchIn(σ_{hps})
Multiplex()

Figure 40: Pseudo code for *ServerContextSwitch*

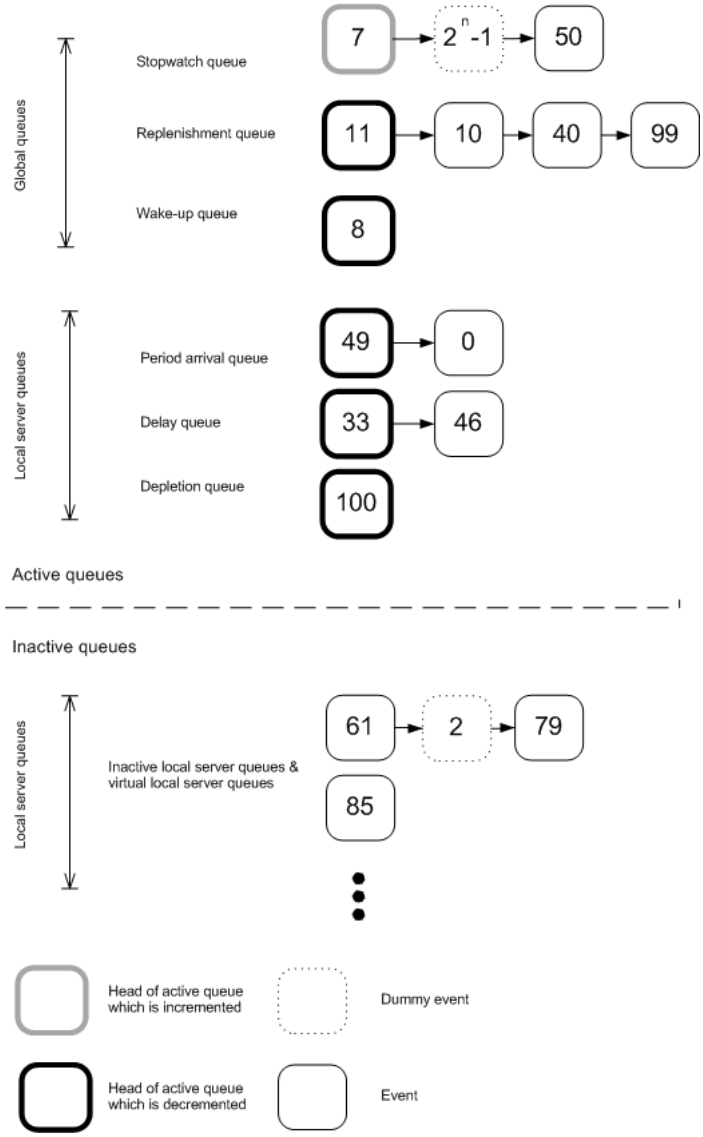


Figure 39: Example of the RELTEQ queues in the system.

if $\sigma_i.readyTasks$ **is** $\emptyset$:
  if $\sigma_i.type$ **is** $PollingServer$:
    $\sigma_i.state = Depleted$
  else:
    $QueueInsert(\sigma_i,\ q_{wakeup},\ FirstNonVirtualEventTime(\sigma_i.ql\ ))$ [(1)]
    $\sigma_i.state = Waiting$
  end if
else
  if $\sigma_i.state$ **is** $Running$:
    $\sigma_i.state = Ready$
  end if
end if
$QueueInsert(\sigma_i,\ q_s,\ 0)$
$ActivateLocalEventQueues(\emptyset)$

Figure 41: Pseudo code for $ServerSwitchOut(\sigma_i)$. [1]This else branch can only be reached by the deferrable server.

QueueRemove(σ_i, q_{wakeup})
SyncEventQueues(σ_i, q_s)
QueueRemove(σ_i, q_s)
ActivateLocalEventQueues($\sigma_i.ql$)
$\sigma_i.state = Running$
if $\sigma_i.readyTasks$ **is** $\emptyset$:
 if $\sigma_i.type$ **is** *PollingServer*:
  $\sigma_i.state = Depleted$
 else
  $\sigma_i.state = Waiting$
 end if
end if

Figure 42: Pseudo code for *ServerSwitchIn*(σ_i)

7.4.5 μC/OS-II Implementation

This section explains how servers and the mapping of tasks to servers are stored inside μC/OS-II, and what the trade-offs of various alternative implementations are.

To describe a server σ_i, we introduce the *Server Control Block* (SCB). It contains information about σ_i's priority, its state, name, and period. The SCB also contains the event queues local to the tasks $\tau_j \in \gamma(\sigma_i)$. To store the SCBs we introduce an SCB table which - similar to the TCB table - is a statically allocated array.

We also need a mapping of tasks to servers, defined by $\gamma(\sigma_i) \subseteq \Gamma$. Γ, the set of periodic tasks in the system, is stored in the TCB table. As explained in section 3.4, the ready state of each task τ in Γ is stored in a table of 8 entries of 8-bits. In this table, the ready state of a task τ_i with priority i is given by bit $i \ mod \ 8$ in the table byte entry $i \ div \ 8$. We define a static mapping $\gamma(\sigma_i) \mapsto \{\tau_{8i}, \tau_{8i+1}, .., \tau_{8(i+1)-1}\}$, where σ_i is a server with priority i, and τ_j is a task in μC/OS-II with priority j. We thus divide the TCB and ready state table in priority bands of 8 tasks, see Figure 43. Conversely, the mapping from a task to a server is given by $\sigma(\tau_j) \mapsto \sigma_{j \ div \ 8}$, where j is the priority of τ_j and i is the priority of σ_i.

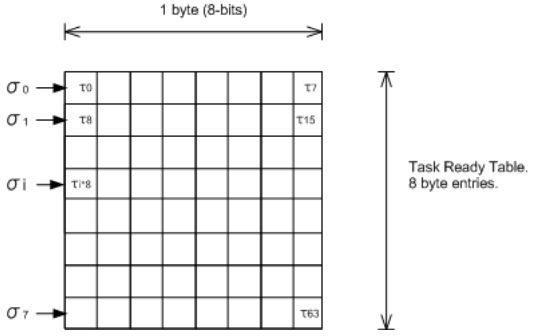


Figure 43: Showing the priority bands in the Ready task table, used to map servers to tasks.

Using this static mapping, no new data structures are introduced for the mapping. Checking if a server σ_i has any ready tasks is done with one $O(1)$ lookup, by inspecting the 8-bit value of entry $i \; div \; 8$ in the ready table, and we can reuse the existing μC/OS-II scheduler as the local scheduler. We also leave the existing μC/OS-II mechanism for changing the ready-state of tasks intact. Finally, every task $\tau_k \in \gamma(\sigma_i)$ has a higher priority than any task $\tau_l \in \gamma(\sigma_j)$, where the server σ_i has a higher priority than σ_j, i.e. directly comparing the priority between two tasks in the system is still possible.

The ready state of tasks is changed by several μC/OS-II functions by directly setting the bits in the ready table. No single function or hook from a function to set the ready state is provided. Whenever μC/OS-II wakes up a task by changing its ready state, the status of the server to which the task belongs is not changed. This is a problem when the server's state is set to *waiting*, and waking up this task should result in the server becoming *ready*. As we assume servers to be independent outside of non-preemptable FPDS subjobs, only sleep and periodic arrival event expirations could wake a server up. However, we want to allow existing applications and future extensions to remain using the existing mechanisms for waking up tasks, so our implementation should handle any corresponding change in the server status. For this reason, to determine if a server is *ready*, we look at its status, but also perform an additional check: whenever its status is *waiting*, but has ready tasks, we change the server's status to ready. This makes the complexity of the server scheduler O(M), with M the number of servers in the system, but we don't have to change the μC/OS-II kernel code at every point a task's ready state is modified. To make the server scheduler $O(1)$, the source code needs to be modified at every point a task's ready state is changed, to change its parent server state accordingly. Then, the ready states of servers can be saved similarily to that of tasks, in a server ready state table where the highest priority ready server can be found with a single bit lookup (as described in 3.4). Because of the static mapping, even without additions to the kernel code, determining the HPT inside a server is still $O(1)$, as only a single byte-entry from the ready table needs to be inspected (as shown in Figure 43). This limits the number of servers to 8, with up to 8 tasks per server. If we want a dynamic number of servers and tasks per server, we need to introduce a task list for each server, and provide methods to create the dynamic mapping, i.e. assign tasks to servers. In this case, we can no longer compare a single byte entry from the ready table to determine the HPT within a server, making the local scheduler $O(N)$. This, again, can be solved by adding additional code to each point where a task's ready state is changed. We could introduce an additional task ready list for each server, which is sorted by priority. Whenever a task's ready state is changed, we update this list accordingly. To determine the HPT within a server we then simply select the head of the ready task list of the server, which is $O(1)$.

To minimize the number of kernel modifications, we decided to use the static mapping. No new task (ready) lists were introduced, no additional code was added around task state changes. 8 servers can be created ($MaxServers$ is 8 in section 7.1), where each server can contain up to 8 tasks ($MaxTasksPerServer$ is also 8).

7.5 Hierarchical Scheduling

In section 2.5 we discussed the four mechanisms required for guaranteeing resource provisions, as identified by Rajkumar et al [02]. In this section we describe how our implementation of 2-level HSF with servers addresses each of them.

7.5.1 Admission

Admission of new subsystems is allowed only during the integration, not at runtime. The admission testing requires analysis for hierarchical systems, which is outside the scope of this thesis.

7.5.2 Monitoring

There are two reasons for monitoring the budget consumption of servers: (i) handle the budget depletion and (ii) allow tasks to track and adapt to the available budget. Budget depletion is handled by the budget depletion event, which we introduced in section 7.4.1. In order to allow other tasks to track the remaining budget β_i of a server σ_i, we introduce new method *ServerBudgetLeft*(σ_i). When the server's state $\sigma_i.state$ is *depleted,* it returns 0. Otherwise, it returns the arrival time of σ_i's depletion event.

ServerBudgetLeft(σ_i). Function which returns the currently remaining budget β_i of a server σ_i. Pre-condition: a server exists at priority i.

7.5.3 Scheduling

In 3.4 we introduced the μC/OS-II scheduler function, *OSSchedNew()*. Our hierarchical scheduler replaces the original scheduler, by placing a hook inside the original scheduler, *OS_SchedNew(),* so that our hierarchical scheduler, *HSFScheduler()*, is executed instead. The hierarchical scheduler consists of two parts: the global and local scheduler. The global scheduler *GlobalScheduler()* returns which server should run next, where *LocalScheduler()* returns which task belonging to this server should run. By using this approach, we can implement different global and local scheduler policies. Our implementation provides a fixed priority scheduler for both the global and local level. The general *HSFScheduler*() and our fixed priority global and local scheduler are shown in figures 44, and 45, and 46, respectively. σ_i refers to a server with priority i, where 0 is the highest priority. If, after the server context switch, the server is not ready to run (e.g. in the case of a depleted polling server, see section 7.4.2), rescheduling is required. The additional ready check in Figure 45, line 2, is added for μC/OS-II compatibility reasons and is explained in section 7.4.5. The actual context switching of tasks is performed by μC/OS-II, after the scheduler, *HSFScheduler()*, has been called and the returned HPT, τ_{hp} is different from the currently running task (Figure 7 in section 3.3 shows this process when calling the scheduler when returning from an interrupt). The operator $[i]$ is used to denote the ith entry of a table.

```
σ_hps= GlobalScheduler()
if σ_hps ≠σ_cur:
  ServerContextSwitch()
  σ_cur =σ_hps
  if σ_cur.state ≠ Running:
    return HSFScheduler()
  end if
end if
τ_hpt = LocalScheduler()
return τ_hpt
```

Figure 44: Pseudo code for hierarchical scheduler, *HSFScheduler()*

```
for i in [1, MaxServers]:
  if σ_i.state is Waiting and if σ_i.readyTasks ≠ ∅:
    σ_i.state = Ready
  end if
  if σ_i.state = Ready:
    return i
  end if
end for
return σ_idle
```

Figure 45: Pseudo code for fixed-priority global scheduler.

```
τ_hpt = σ_cur.priority *8 + OSUnmapTbl[OSRdyTbl[σ_cur.priority]]
return τ_hpt
```

Figure 46: Pseudo code for fixed-priority local scheduler

7.5.4 Enforcement

When the server has used up its budget and the depletion event expires, its currently running task is preempted and the server is switched out. This is possible since we assume servers to be independent outside of FPDS jobs (which we deal with in the next section).

7.6 H-FPDS

In Chapter 4 we extended μC/OS-II with FPDS, to reduce the cost of context switches. During the subjob execution of an FPDS task, preemption of the task is deferred until the next preemption point. With the introduction of servers with associated budgets, an FPDS task can be running at the point the server's budget becomes depleted. As FPDS can also be used as a guard around a critical section (see section 2.3.3), the FPDS task may not be preemptable at the point of depletion. In order to deal with this problem, we want to extend FPDS to work with reservations and hierarchical scheduling. We will call this extension H-FPDS, to distinguish it from the normal FPDS implementation described in Chapter 4.

For H-FPDS we identify two modes: *local* and *global H-FPDS*. When a task is scheduled according to local H-FPDS, H-FPDS only holds relative to other tasks in the same subsystem. When executing a local H-FPDS task, the selection of a new task is thus only skipped in the local scheduler, and any task from a higher priority server may still preempt it. Only *local resources,* i.e. resources shared only among tasks in the same server, are allowed to be accessed during a local H-FPDS subjob. This ensures that a higher priority server cannot be blocked on a resource which was acquired by the preempted local H-FPDS task. Whenever the server σ_i is depleted during the execution of a local H-FPDS task τ_j, σ_i is switched out, which causes τ_j to be preempted. When σ_i is switched back in, τ_j task must be allowed to continue its execution, at least until the next preemption point. This way no other higher priority task in the server σ_i can block on the resource that τ_j acquired. The requirement of only allowing a local H-FPDS task to access local resources can be made explicit by adding checks to the lock functions of shared resources (such as a lock mutex function) that the requested resource has been created (is owned) by a task of the currently running server.

Under global H-FPDS, a running global H-FPDS task τ_j cannot be preempted by any other task, either within the current server σ_i or a higher priority server. We can thereby reduce the cost of server context switches, similarly to that of task context switches with FPDS. It can also be used to guard critical sections with access to *global resources*, i.e. resources shared amongst multiple subsystems. In this case, depletion of the budget of the server σ_i of task τ_j is a problem, as we can no longer simply switch out σ_i: if τ_j has access to a global resource shared resource on which another server depends, it could cause very long blocking periods. This problem also occurs in resource access protocols which are used for hierarchical systems, such as HSRP [38] and SIRAP [39]. To prevent depletion of the budget in a server while a global resource is accessed, HSRP suggests an *overrun* mechanism (with or without *payback*). When the server budget becomes depleted during the execution of a critical section, the budget is temporarily increased with the *server overrun time*, the longest time a task in that server is allowed to spend in a critical secction. If *payback* is used, when an overrun occurs, the used overrun time is substracted from the server's budget at the next replenishment. SIRAP, on the other hand, uses a *skipping* mechanism: a critical section can only be entered when the server's budget is larger than or equal to the worst-case execution time of the critical section.

Global H-FPDS can always be used, regardless of which type of resources need to be accessed, but cannot be preempted by higher priority servers, and requires an overrun or skipping mechanism. With local H-FPDS, higher priority servers may still preempt the server, but only local resources can be used. When we want to use global H-FPDS to prevent server context switches, but do not use any global resources, we could use a *hybrid* approach. As only local resources can be accessed with hybrid H-FPDS, the server can be switched out safely when it is depleted. This prevents the need for a mechanism of dealing with server depletions within a critical section. Like local H-FPDS, we need to make sure the hybrid H-FPDS task is allowed to continue when a server is switched back in, and that no global resources are acquired. The different options are summarized in Table 9.

We assume H-FPDS tasks should be able to access global resources, such as the DMA in the camera platform we described in Chapter 1. We therefore

implemented global H-FPDS. According to [18], whether to use the overrun or skipping mechanism heavily depends on the parameters of the system, and decide to support both for this reason. To this end we extend FPDS with *conditional preemption points* for skipping, and extend the depletion and replenishment events with support for overruns.

	τ_{cur} can be preempted by			τ_{cur} can lock resources	
	Depletion of σ_{cur}	$\tau_{hp} \in \gamma(\sigma_{cur})$	$\tau_j \in \gamma(\sigma_{hp})$	Locally	Globally
Local	Yes	No	Yes	Yes	No
Global	No	No	No	Yes	Yes
Hybrid	*Yes*	*No*	*No*	*Yes*	*No*

Table 9: Two H-FPDS modes Local & Global, with a third *Hybrid* alternative. σ_{cur} is the currently executing server, τ_{hp} is a task with a higher priority than the currently running task, τ_{cur} the currently running task, σ_{hp} is a server with a higher priority than σ_{cur}.

7.6.1 Interface

H-FPDS support is offered to the application with the interface described below. To schedule a task according to global H-FPDS, *SetFPDS()* is used, as introduced in section 4.1.

SetHFPDS(σ_i, X_s, *payback)*. Procedure to enable global H-FPDS tasks to be used in server σ_i, and specify their behavior. X_s is the maximum *server overrun time* in time units, with $0 \leq X_s < \Theta_i$. When X_s is 0, the overrun mechanism is disabled for σ_i. In this case, skipping must be used to prevent budget depletions inside an H-FPDS subjob. The *payback* flag is used to enable or disable the payback mechanism. Pre-condition: a server exists at priority i. Post-condition: $\sigma_i.X_s = X_s$, $\sigma_i.PaybackEnabled = payback$.

ConditionalPreemptionPoint(C_ι, *allow_yield)*. Procedure which is called by a global H-FPDS task τ_{cur} in server σ_{cur} to define a conditional preemption point in its subjob. This is used to implement skipping. C_ι defines the worst-case execution time of the next subjob in time units, with $0 < C_\iota \leq \Theta_i$. Pre-condition: τ_{cur} is a global H-FPDS task, set by SetFPDS(τ_{cur}). Post-condition: When C_ι is larger than the remaining budget β_{cur} of σ_{cur}, the task is blocked (and thus preempted) until enough budget becomes available and it is allowed to run again. When enough budget is available, calls - and returns the value of - *PreemptionPoint*(*allow_yield*).

7.6.2 Scheduler

In figure *13*, we showed the FPDS scheduler based on the *OriginalScheduler()* function, which is the original scheduler in μC/OS-II. In 7.5.3, we replaced the *OriginalScheduler()* with the hierarchical scheduler, *HSFScheduler()*. During the execution of a global H-FPDS task, selecting a new server and task should both be skipped. We can thus leave the FPDS scheduler unchanged.

In the *PreemptionPoint(allow_yield)* procedure in Chapter 4, the original μC/OS-II scheduler was called to determine if a new task was ready. To implement optional preemption points, the context switch is only performed if the *allow_yield* parameter is *True*. As *HSFScheduler*() already switches the server context when a higher priority server is available, we cannot directly call this procedure when *allow_yield* is *False*. Instead, we call the *GlobalScheduler()* to determine the highest priority ready server. When the highest priority ready server $\sigma_{hp} \neq \sigma_{cur}$ (σ_{cur} being the current server), we return *True*. As the server context switch, synchronization and *LocalScheduler()* have not yet been run, the *ShouldSchedule* flag remains set. When $\sigma_{hp} = \sigma_{cur}$, we call *LocalScheduler()* to determine if a new higher priority task within σ_{cur} is available, and reset *ShouldSchedule*. When *allow_yield* is *True*, we call *HSFScheduler()* and yield if needed.

```
PreemptionPoint(allow_yield):
if allow_yield is True:
  if ShouldSchedule is True:
    ShouldSchedule = False
    τhp = HSFScheduler()
  end if
  if τhp ≠ τcur:
    ContextSwitch(τcur,τhp)
    return True
  end if
  return False
else:
  if ShouldSchedule is True:
    σhp = GlobalScheduler()
    if σhp ≠ σcur:
      return True
    end if
    ShouldSchedule = False
    τhp = LocalScheduler()
  end if
  return τhp ≠ τcur
end if
```

Figure 47: Pseudo code for H-FPDS preemption point.

7.6.3 Skipping

To implement skipping, we implement conditional preemption points. Whenever a preemption point is reached, the H-FPDS task calls the conditional preemption point procedure and specifies the worst-case execution time of the next subjob, C_ι. The procedure checks if the current server's budget β_i is large enough to allow the H-FPDS subjob to finish before the server is depleted. When $\beta_i < C_\iota$, the procedure will wait for the server's next replenishment period until $\beta_i \geq C_\iota$, causing the task to be preempted. When $\beta_i \geq C_\iota$, the normal preemption point procedure is continued, as described in Chapter 4. When the task wants to

alter its execution path based on the available budget, the interface described in section 7.5.2 can be used to determine the available budget. Note that the time needed to return to executing the subjob after determining enough budget is available must be included in C_ι, to avoid race conditions.

To wait for the server's replenishment period, we extend the reservations interface with the following procedure:

WaitForNextServerPeriod(). Procedure to wait for the next server period (replenishment) of server σ_{cur}, where σ_{cur} is the currently running server.

To implement the procedure, we extend the SCB with a server period semaphore, which - similar to the semaphore for a periodic task arrival - is used to wake up any tasks that were waiting for the next server period. *WaitForNextServerPeriod()* blocks on this semaphore, which is reset on every server replenishment by the replenishment handler (see section 7.4.1).

Pseudo code for conditional preemption point is shown in Figure 48.

ConditionalPreemptionPoint(C_ι, allow_yield):
while *ServerBudgetLeft*(σ_{cur}) $< C_\iota$:
  WaitForNextServerPeriod()
end while
return *PreemptionPoint(allow_yield)*

Figure 48: Pseudo code for optional preemption point. σ_{cur} is the currently running server.

7.6.4 Overrun & Payback

To support overrun, we introduce several new attributes to each server: (i) the maximum overrun time, X_s, (ii) a flag *PaybackEnabled*, which indicates if the payback mechanism is enabled for this server, and (iii) a flag *InOverrun*, which is set if the server is currently consuming overrun budget. We modify the depletion and replenishment event handlers to deal with overrun.

Whenever the server has depleted its normal budget, and the currently running task is in global H-FPDS mode, the *InOverrun* flag is set. The budget β_i is then increased by the overrun budget, X_s, by inserting a new virtual depletion event with expiration time X_s.

When the server's replenishment event is due, we first check for the *InOverrun* flag. If the *PaybackEnabled* flag is set, we establish β_o: how much overrun budget has been consumed (which is equal to $X_s - \beta_i$, where β_i is determined using *ServerBudgetLeft()*, see section 7.5.2). Otherwise, β_o is zero. We then reset the *InOverrun* flag. The server's budget is replenished as usual, but β_o is substracted from the new budget.

When a preemption point is reached and *PreemptionPoint()* is called, the procedure will check for the *InOverrun* flag. If it is set, we set the server's state to *depleted*, and yield.

The replenishment, depletion and preemption point functions are described by the pseudo code in figures 49, 50, and 51, respectively.

When the depletion event is handled and the *InOverrun* flag is already set, this means the overrun budget has been depleted. This is considered a temporal

Replenishment Event(σ_i):
β_o =0
if $\sigma_i.InOverrun$:
 if $\sigma_i.PaybackEnabled$:
  $\beta_o = \sigma_i.X_s - ServerBudgetLeft(\sigma_i)$
 end if
 $\sigma_i.InOverrun = False$
end if
if $\Theta_i - \beta_o > 0$:
 $QueueInsert(\sigma_i, q_{depletion}, \Theta_i - \beta_o)$
 σ_i.state = *Ready*
end if
$QueueInsert(\sigma_i, q_{replenishment}, \Pi_i)$

Figure 49: Pseudo code replenishment event handler

Depletion Event(σ_i):
if $\tau_{cur}.SchedulePolicy$ **is** FPDS:
 $\sigma_i.InOverrun = True$
 $QueueInsert(\sigma_i, q_{depletion}, \sigma_i.X_s)$
else
 σ_i.state = *Depleted*
end if

Figure 50: Pseudo code depletion event handler

if $\sigma_{cur}.InOverrun$:
 σ_{cur}.state = *Depleted*
 τ_{hp} =*HSFScheduler()*
 $ContextSwitch(\tau_{cur}, \tau_{hp})$
 return *True*
end if
PreemptionPoint()

Figure 51: Pseudo code of overrun check, executed at the start of the preemption point procedure.

fault, and can be handled by e.g. rolling back a transaction created when the subjob was started. Fault handling is outside the scope of this thesis, and we simply deplete the server.

7.7 Evaluation

In the following sections we will evaluate our reservations extensions. In section 7.7.1 we demonstrate the behavior of the different server types, and H-FPDS. Section 7.7.2 shows how many lines of code were added in which components, to implement the functionality introduced in this Chapter. The memory footprint of the structures used for reservations is shown in 7.7.3. Finally, the overhead is discussed in section 7.7.4.

7.7.1 Behavior

Figure 52 shows the trace of an example application consisting of three servers, each of a different type and with one or more tasks. The parameters of the tasks and servers, which are shown in Table 10, were chosen such that it demonstrates the behavior specific to each type of server [2]. The deferrable server consists of three tasks, which we denote as τ_{DS1}, τ_{DS2}, τ_{DS3}. The polling server consists of one task, τ_{Poll1}, and periodic server has the task τ_{PS1} assigned to it. Note that the periodic server also always contains an idle task, which is denoted here as $\tau_{PS-Idle}$. Underneath the execution traces of the tasks, Figure 52 also shows the capacities of the servers.

Server	Deferrable Server			Polling Server	Periodic Server
i	1			2	3
Θ	20			10	15
Π	50			50	50
Task	DS1	DS2	DS3	Poll1	PS1
j	1	2	3	1	1
ϕ	0	10	40	0	0
C	5	5	15	7	10
T	50	50	50	200	50

Table 10: Parameters of task τ_j in server σ_i, with j and i the priority of the task and server, respectively.

In the example trace, at execution time t = 0 (marked with number 1 in the figure), τ_{PS1} is ready but tasks in the higher priority deferrable server are allowed to run first: τ_{DS1} is ready and activated. At t = 5 (2), no more tasks are ready in the deferrable server, and the polling server is switched in. Part of the poller job of τ_{Poll1} is executed, until a wake-up event occurs at t = 10 (3). The deferrable server is switched in, the periodic arrival event for τ_{DS2} is handled, and τ_{DS2} is activated. After this, execution of the polling job is resumed at t = 15 (4). When the job is finished, the workload of the polling server is exhausted, and the budget is depleted at time t = 17 (5). The periodic server is still in the ready state, and is now switched in, after which τ_{PS1} is activated. At t = 27 (6), no more tasks are ready in the periodic server, and its remaining budget is idled away by $\tau_{PS-Idle}$. The server is depleted at t = 32 (7), and it is switched out. At this point, no servers are ready and the system idle server becomes active. At t = 40, another wakeup of the deferrable server occurs (8), as the first period for task τ_{DS3} arrives. As C_{DS3} is 15, and ony a budget of 10 remains at this point, τ_{DS3}'s job cannot be finished in this server period. At t = 50 (9) all servers are replenished, and a new instance of τ_{DS1} arrives. The local scheduler of the deferrable server selects τ_{DS1} over τ_{DS3}, and τ_{DS3} needs to wait until the job of τ_{DS1} is finished. At t = 65 (10), no more tasks are ready in the deferrable server, and the global scheduler selects the polling server. While switching in the polling server, it is established no tasks are ready, and the server's budget is depleted immediately. At t = 50, a new period arrived for τ_{PS1}. Handling the period event, however, is deferred until t

[2] For this example, the utilization of the tasks is higher than that of the server, and is thus not schedulable.

= 65 (11), when the periodic server becomes the highest priority ready server, and its queues are synchronized. This again activates τ_{PS1}.

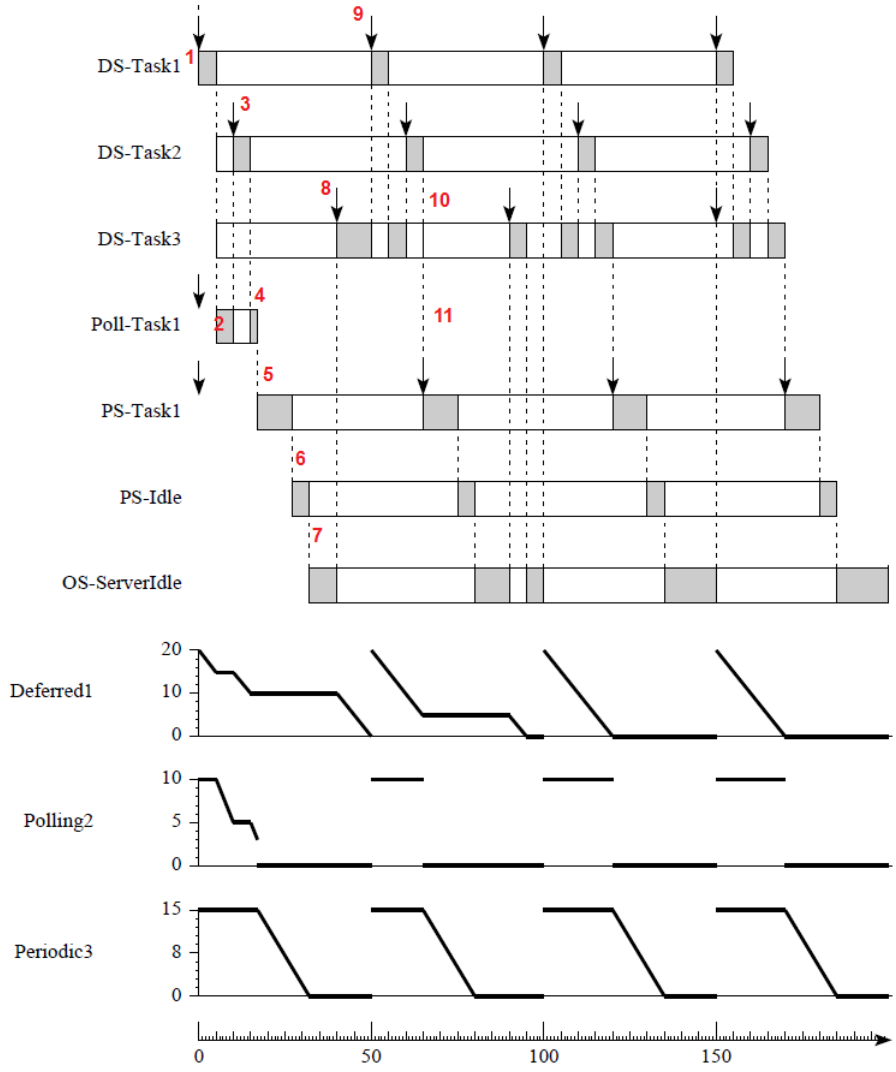


Figure 52: Trace of example application with parameters as shown in table 11. The numbers in the traces correspond to the task and server activations as explained in the text.

H-FPDS - Skipping

Figure 53 shows the behavior of an example application with a deferrable server and a periodic server. The periodic server contains two tasks, τ_{PS1} and τ_{PS2}, where τ_{PS1} is scheduled according to H-FPDS, using skipping. τ_{PS1} consists of two subjobs. The parameters of the application are shown in Table 11.a. At t = 4 (1), the first subjob of τ_{PS1} is activated. A conditional preemption

point occurs after the first subjob of τ_{PS1} is finished, at t = 11 (2). At this time τ_{DS1} is ready, and control is yielded to this task. At t = 15, control is returned to the periodic server. The second subjob of τ_{PS1} requires 9 time units to execute, but the remaining budget is only 5: τ_{PS1} thus waits for the next replenishment and is *skipped* (3). Although τ_{PS1} is waiting, the periodic server still has remaining budget, which is used to execute τ_{PS2} (3) [3]. After replenishment of the servers, the task in the higher priority server is allowed to run first (4). When the periodic server is switched back in, τ_{PS1} is finally activated, and the skipped job is executed (5).

	Deferrable Server	Periodic Server	
i	1	2	
Θ	8	12	
Π	20	20	
Tasks	DS1	PS1	PS2
j	1	1	2
T	10	40	20
ϕ	0	0	0
$C_{i,k,l}$	4	7 + 9	3

(a)

	Deferrable Server	Periodic Server
i	1	2
Θ	8	20
Π	50	50
X_s	-	9

Tasks	DS1	PS1
l	1	1
T	10	100
ϕ	0	0
$C_{i,k,j}$	4	7 + 9 + 9

(b)

Table 11: (a) Task and server parameters of example application, with trace in Figure 53. τ_{PS1}consists of two subjobs, and is scheduled with H-FPDS. (b) Parameters of example application with traces in Figure 54.

H-FPDS Overrun

Fig 54 shows the behavior of an example application, with parameters shown in table 11.b. The periodic server consists of a task τ_{PS1}, which consists of three subjobs, is scheduled with H-FPDS using the overrun mechanism. In 54.a, payback is disabled. At t = 15 (1), τ_{PS2} is activated, and starts executing the second subjob. After 9 more units (t = 24), when the second subjob is finished, the optional preemption procedure is called. As no other task is ready, τ_{PS2} continues with its third subjob, with a computation time of 9 time units. At this time (t = 24), however, only 4 time units of budget are left. This causes the budget to be depleted at time t = 28 (2). The budget is then replenished with the overrun budget X_s. 5 time units later, at t = 33 (3), the job is finished, and calls the preemption point procedure. The remaining overrun budget is then discarded. At t = 50, the servers are replenished, and normal execution continues. Fig 54.b. shows the same test application, now with payback enabled. At t = 50 (1), the overrun budget that was consumed (5 units), is deducted from the replenishment budget. Finally, if the normal budget replenishment would occur during an overrun (for example, if the server period would have been 30 instead of 50), the server would be replenished in the same way as described

[3] By allowing another task in the server to run while the H-FPDS job is waiting (being skipped), the protocol cannot be used to share a single stack, like the SRP-based SIRAP.

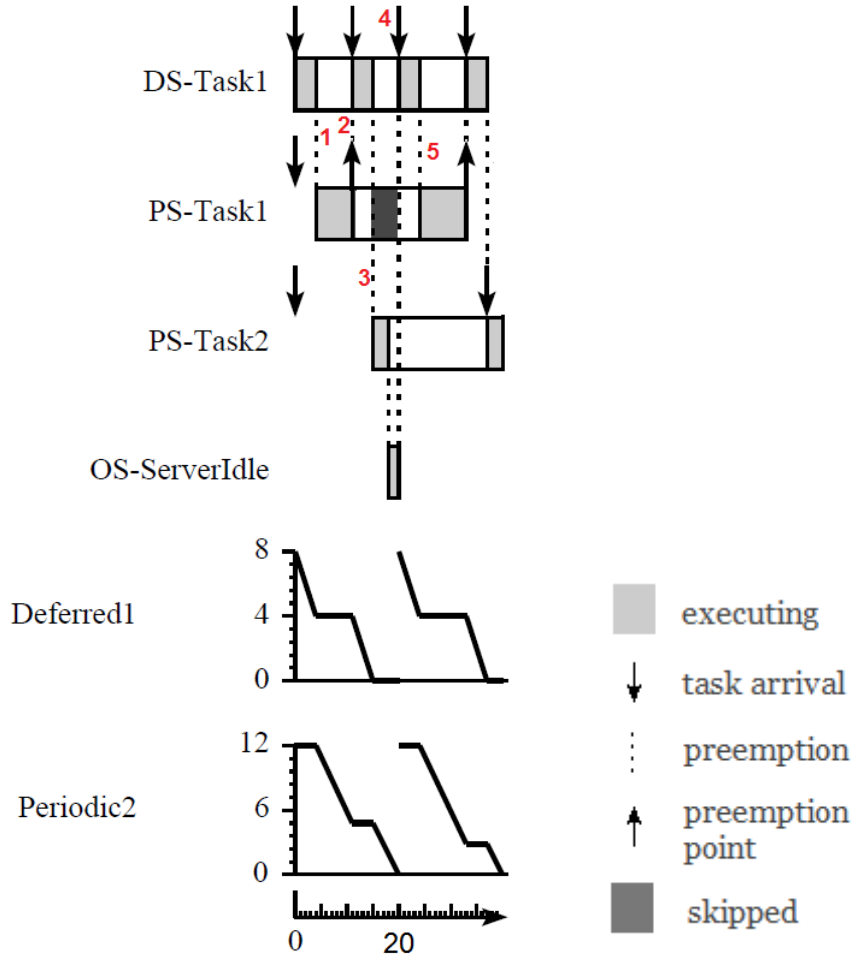


Figure 53: Example behavior of H-FPDS with skipping

above, and the overrun would stop (i.e. the overrun flag is reset).

7.7.2 Modularity

The implementation of reservations consists of the RELTEQ extension, HSF and servers, and H-FPDS. The RELTEQ extension is 454 lines of code, and is added to the RELTEQ module OS_TEV_Q.C. The reservation interface and HSF is implemented in a separate module OS_TEV_R.C, and is 332 lines of code. H-FPDS support is added to the overrun and replenishment event handlers (34 lines), and to the FPDS module OS_TEV_F.C. 79 lines were added to the header file, OS_TEV.H. None of the original μC/OS-II code was modified or deleted. 2 lines were added to to OS_CORE.C, to initialize the idle server and to start the highest priority ready server on initialization. To provide a hook for the HSF scheduler in the original *OS_SchedNew()* procedure, another 3 lines were added. Reservations (including the RELTEQ extension, HSF, servers and H-FPDS) can be disabled or enabled with a single compiler directive. Table 12

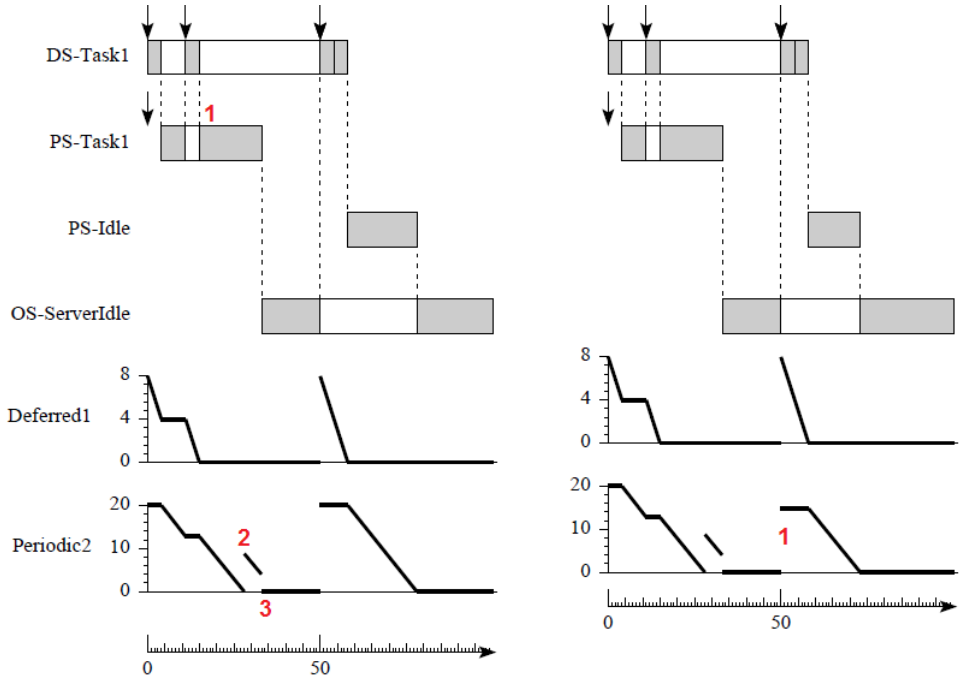


Figure 54: Overrun (a) without payback, (b) with payback.

provides an overview of the all the modules introduced in this thesis.

	Time Management	Reservations		Periodic Tasks
File	OS_TEV_Q.C	OS_TEV_Q.C	OS_TEV_R.C	OS_TEV_R.C
Contains	RELTEQ	Extension	HSF, Interface	Interface
LOC	454	180	332	94
	FPDS	H-FPDS		Base
File	OS_TEV_F.C	OS_TEV_F.C	OS_TEV_Q.C	OS_TEV.H/.C
Contains	Interface	Interface	Overrun, Replenish	Defines/Initialization
LOC	80	40	34	171/55

Table 12: Overview of file modules and the lines of code, which totals 1440 lines in 6 files.

7.7.3 Memory footprint

As described in section 7.3.4, the queue structure (see 5.4.2) is extended with the flag IsVirtual. We also add a flag IsActive which can be used by procedures to determine if the queue is active, without having to inspect the active queues list. Note that as an additional optimization, IsVirtual and IsActive could also be stored in the highest bits of EventType. In 7.4.5, we introduced the SCB and the SCB table. As an optimization for the scheduler implementation,

we add a global pointer (SCBCur) to the currently activated server's SCB. The priorities of the currently activated server σ_{cur}, and the highest priorty server, σ_{hp} are also stored in a global variables. In section 7.3.1, we describe the process of activating queues, which involves setting the next pointer of the last active event queue, which is pointed to by LastActiveEventQueue. The size of the semaphore structure and name text strings (denoted as |semaphore| and |name|) are variable in μC/OS-II, depending on the configuration settings.

Field	Memory (bytes)
Budget	4
Period	4
Type	1
Priority	1
State	1
InOverrun	1
PaybackEnabled	1
LocalQueues	3 * 9
PeriodSemaphore	|Semaphore|
ServerName	|Name|
Total	40 + |Name| + |Semaphore|

(a)

Field	Memory (bytes)
EventFirst	2
EventType	1
QueueNext	4
ISVirtual	1
ISActive	1
Total	9

(b)

Global Variable	Memory (bytes)
SCBCur	4
PrioServCur	1
PrioServHigh	1
SCBTable	$MaxServers$ * |SCB|
LastActiveEventQueue	4
GlobalQueues	2 * 9
StopwatchQueue	9
Total	37 + $MaxServers$ * |SCB|

(c)

Table 13: Memory footprint of (a) SCB structure, (b) queue structure, and (c) global variables.

7.7.4 Processor Overhead

The additional overhead of reservations is caused by global scheduling, server context switching and handling server events. We list the complexity of these below. We denote the maximum number of tasks in the system as N, the maximum number of servers as M, and the maximum number of tasks per server L.

Replenishment and Depletion. Exactly once every server period a server is replenished (we consider H-FPDS separately). When the server is replenished, a new event is inserted in the replenishment event queue. As the number of events in this queue is lineair in the number of servers, inserting the replenishment event is $O(M)$. At most once every server period, a depletion event is handled. Since the depletion events are stored in a queue local to the server,

and since there is only one event in that queue, the overhead is $O(1)$.

Scheduler. As we traverse the list of servers (see section 7.4.5) to determine the highest priority ready server, the global scheduler is $O(M)$. As we use the original $O(1)$ μC/OS-II approach to schedule tasks locally (see 7.4.5 and 3.4), the additional overhead of the HSF scheduler consists of global scheduling only.

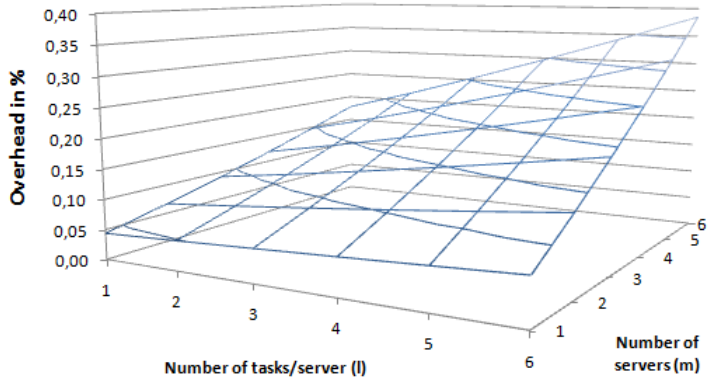
Server Context Switching. Upon a server context switch, all local non-virtual event queues are synchronized. As shown in Fig. 39, two local event queues need to be activated and synchronized. The number of events to synchronize depend on the number of tasks in the server, and is thus $O(L)$. The number of events in the stopwatch queue is lineair in the number of servers, which makes inspecting the stopwatch queue $O(M)$. When switched out, a deferrable server can insert a wake up event, which is also $O(M)$. The complexity of the server context switch is therefore $O(M + L)$.

Handling the local events while synchronizing introduces no additional overhead introduced by using reservations: without reservations a due event of any task would have been handled in the tick handler when it is due. With synchronization, we not only prevent interference from inactive servers, but also batch the handling of events of an inactive server, which can reduce cost to handle an event. Consider server σ_i, which has been inactive for t_o time units, in which ϵ of its local events were due. Without reservations, in the worst case, the tick handler would have been executed ϵ times (which includes traversing the active queues, and multiplexing). With synchronization, traversing the queues and multiplexing is done only once to handle ϵ events.

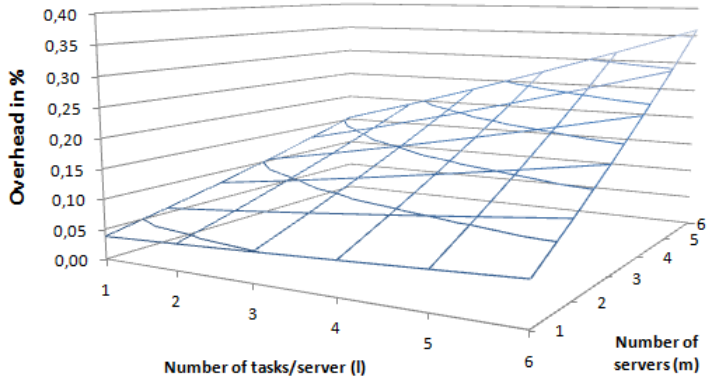
Tick Handler. The tick handler synchronizes all active event queues with the current time. In section 5.4.1, we showed the tick handler to be lineair in the number of due events. To compare the tick handler with and without reservations, we consider an event to arrive every tick (i.e. $\delta = 1$). Without reservations, the tick handler is then $O(N)$. When reservations are enabled, there are two global server queues (replenishment and wake-up queues) whose size is proportional to M. The size of the three local queues is proportional to L. Using reservations, the tick handler is therefore $O(M + L)$.

To show the difference in overhead between reservations and no reservations, and the deferrable server and a periodic server, we measured the overheads on a group of different test task sets. The number of tasks N in a set depend on the parameters l and m. When reservations are enabled, m servers are created with l tasks each (i.e. $M = m$, $L = l$). When reservations are disabled, $l * m$ tasks are created. In both cases, $N = l * m$. Each task in a set is periodic, with a computation time C of 50ms (5 ticks). The capacity of each server is set to $L * 50$ms, and the period to 2000ms (200 ticks). Each task set is executed for 2000ms. The task period is set to a value larger than 2000ms. During a test run, exactly one instance of every task is executed, and each server will be replenished and depleted exactly once. To include overhead of wake-up events for deferrable servers, the tasks are released after each other, with the highest priority task ($j = 1$) in the highest priorty server ($i = 1$) released last, i.e. ϕ_j (in ticks) of a task j in $\gamma(\sigma_i)$ is $200 - (i-1)lC - jC$. This causes each deferrable server to be woken up once during its server period.

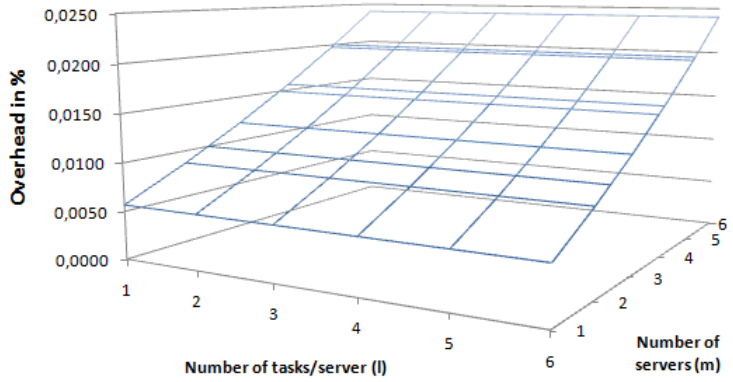
Fig. 55.a shows the overhead of the task set using deferrable servers expressed in the percentage of the total runtime (200 ticks). The overhead includes the overhead of the RELTEQ tick handler (including handling all events), and the global scheduler, including server context switching, but without local scheduling (for which the original μC/OS-II scheduling mechanism is used). Fig. 55.b shows the overhead when periodic servers are used instead of a deferrable server. The overhead difference between the two server types, shown in Fig. 55.c, is caused by the handling of the wake-up events for deferrable servers. We also measured the overhead of running the task set without servers (i.e. with $n * m$ periodic tasks), consisting of the RELTEQ tick handling (including handling all events), and again expressed it in the percentage of the total runtime. The difference between the overhead with and without reservations is shown in 55.d. It was obtained by subtracting the overhead percentages of the task set without reservations, from the overhead percentages of the deferrable server set (as shown in 55.a). The overhead with reservations is higher because of the additional cost of global scheduling. As noted above, the overhead of tick handling with and without reservations is $O(M + L)$ and $O(N)$, respectively. As $N = m * l$, $M = m$, $L = l$, the overhead of the tick handler without reservations is $O(m * l)$ compared to $O(m + l)$ with reservations, which causes less overhead for higher values of l and m. Fig. 56 shows the difference in overhead between reservations (with deferrable servers) and no reservations, this time only taking into account the tick handler (i.e. we leave out the overhead caused by global scheduling and server context-switching). Looking only at the tick handler, we can actually observe less overhead for the reservations task set.



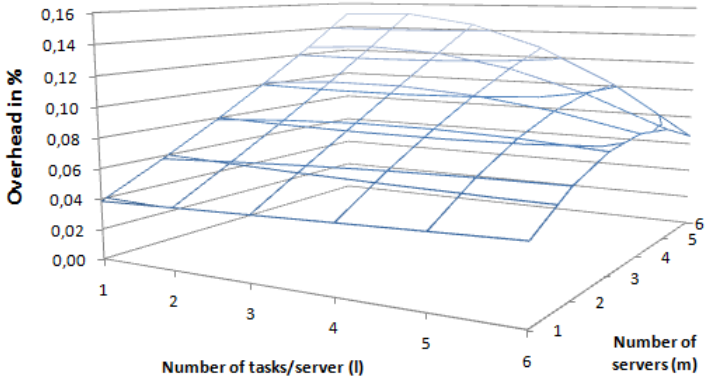
(a)



(b)

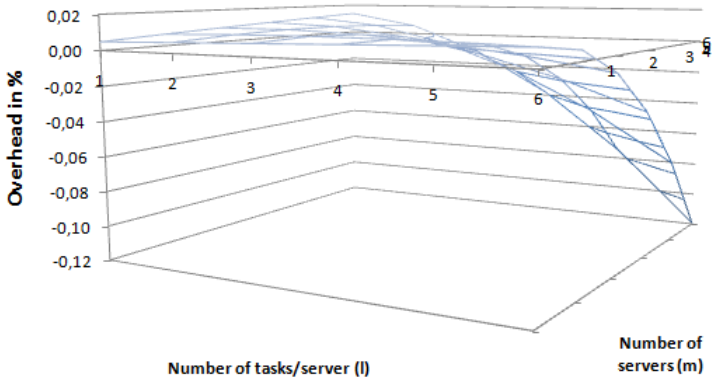


(c)



(d)

Figure 55: Overhead % (a) of reservations with deferrable server, (b) of reservations with periodic server, (c) difference between deferrable and periodic server, (d) difference between reservations and no reservations,



(e)

Figure 56: Difference of runtime overhead % between tick handler using reservations and without reservations.

To support H-FPDS, we implemented two mechanisms to prevent depletion during the execution of a H-FPDS task: skipping & overrun. We discuss the overhead introduced by these methods, and show the measurements in table 14.

Skipping. The skipping mechanism is implemented with a conditional preemption point procedure, which performs an additional check for sufficient remaining budget. To check for sufficient budget, *ServerBudgetLeft()* is called to get the currently remaining budget. In the best case, enough budget is remaining, and no further action is taken. Otherwise, the task is blocked until replenishment by calling *WaitForNextServerPeriod()*, after which the budget check is repeated. We measured the overhead of the conditional preemption point (C_{skip}), where n_r is the number of replenishments needed before enough budget is available.

Overrun. In the best case, no overrun occurs and the overhead is 0. The overhead of an overrun consists of the depletion event handler ($C_{overrun}$), which replenishes the budget to X_s. This is done by inserting an event into the replenishment event queue, which is $O(M)$. When payback is enabled, the replenishment event needs to calculate the used overrun budget (at the cost of $C_{payback}$). Finally, the overrun is ended by either a replenishment, or calling a preemption point, which reset the overrun status (C_{dyield}) and deplete the budget ($C_{nopayback}$), respectively.

Although skipping is simple in terms of implementation parts, the overhead of waiting for a next server period is relatively high compared to the overhead for the different mechanisms needed for overrun. This is because our implementation uses a relatively high-level OS primitive, namely semaphores. For overrun, when payback is disabled, only the overrun flag needs to be reset. When pay-

back is enabled, the consumed overrun budget must be calculated and a call to *ServerBudgetLeft*() is needed.

	Skipping	Overrun			
	C_{skip}	$C_{overrun}$	C_{dyield}	$C_{payback}$	$C_{nopayback}$
Cycles	64 + 3876 * n_r	2038	114	448	34
μs	0.6 + 17.2 * n_r	20.4	1.1	4.5	0.3

Table 14: Overhead H-FPDS, for $M = 8$

8 Conclusion

In this thesis we have presented an efficient, extensible and modular design for extending a real-time operating system with periodic tasks, FPDS, and reservations. It relies on RELTEQ, a component to manage timed events in embedded operating systems. We have successfully implemented our design in the μC/OS-II real-time operating system, while minimizing modifications to the kernel.

In section 2.5, we described the mechanisms required to implement processor reservations as described by Rajkumar [1]: *admission*, *scheduling*, *enforcement*, and *accounting*. While admission is done offline, we use a two-level HSF (see section 2.6) for scheduling and servers (2.4) for accounting and enforcement. We provide support for deferrable, periodic and polling servers.

The implementation of FPDS is based on a combination of FPNS and preemption points. To support the periodic task model, we also extended μC/OS-II with periodic task primitives. As an FPDS subjob cannot be preempted, when FPDS is used in combination with reservations, some mechanism is required to prevent depletion of the server budget while the subjob is executing. We base our solution, called H-FPDS, on two mechanisms used by other hierarchical synchronization protocols: skipping [39] and overrun [38].

We have identified support for timed event management as an important aspect of the design, as it is required for managing server replenishments, depletions, and periodic task arrivals. Virtual timers are also required, to handle server events which are relative to its consumed budget (such as depletion events). Furthermore, as processor reservations should provide temporal isolation between reserves to guarantee budgets, a mechanism is needed to limit the interference of handling timed events from currently inactive servers. As μC/OS-II only provides delay events, we propose the design of RELTEQ, an efficient and versatile timed event manager. It stores event times relative to each other in event queues, which allows for a low memory footprint, long event interarrival time, and prevents drift from calculating next expiration times. To avoid interference from local events of inactive servers, whenever a server is switched out, its event queues are deactivated, and handling of its events is deferred until the server is switched back in. This method is also exploited to provide efficient virtual timers, i.e. their expiration times need not be recalculated when a server is switched back in. Deactivating event queues of waiting deferrable servers was made possible by using a wakeup events, which wake up the server upon the expiration of a local event which should cause the server to become ready again. We successfully implemented RELTEQ in μC/OS-II, to replace and extend the timer functionality, conforming to the requirements we set in section 5.1.

We evaluated our implementation by looking at the modularity, efficiency and behavior. In order to perform accurate measurements based on execution times in clock cycles, we introduced a module to insert profiling points in the kernel (see section 3.6). We then tested our extensions using the OpenRISC 1000 simulator, to get a general overview of the overhead of various components. It showed that our extension exhibited acceptable overhead, and that in most cases using RELTEQ is more efficient than μC/OS-II' task timers mechanism. In total, our extension consists of 1440 lines of code (excluding comments and blank lines, including compiler directives). An additional 40 lines of code were added to the original μC/OS-II kernel, to provide a hook for the scheduler, extend existing structures, and to replace the existing timer management. Dif-

ferent parts of our extension can be switched on and off at compile time, using three compiler directives. As shown in section 7.7.2, the largest component is RELTEQ (with 44% of the amount of code), but its functionality is used by the other components, reducing the size of their implementations (e.g. the reservation interface and HSF requires 332 lines of code). Finally, the size of the compiled binary image was increased by 31% in total (see 15 in Appendix B).

At the start of this project, we did not expect the design of a timer management component to play such an important role. Starting with the design of RELTEQ turned out to be a good decision, and allowed for a modular and concise implementation of reservations and H-FPDS on top of it. We expect our approach can be useful as a basis for further extensions and implementations of real-time embedded operating systems (see *Future work* below). Finally, we would also like to point out that μC/OS-II' clear and concise structure and good documentation made it very nice to use as a platform for implementing our own extensions.

Future Work

Below we suggest some topics for future work in the field of resource reservations, hierarchical scheduling, H-FPDS, and the improvement of RELTEQ.

Analysis. Shin & Lee [27] describe the schedulability analysis for a HSF with EDF and FP scheduling algorithms, based on the periodic resource model. In the periodic resource model a resource Γ_i is specificied by a pair (Π_i, Θ_i), with Π_i its replenishment period and Θ_i its capacity, like the deferrable and periodic servers. It would be interesting to use and/or extend the analysis to take into account the overhead of HSF implementations.

Improvements on our design. Exploring ideas how our design can be improved might be an interesting future work. In particular improvements to RELTEQ, which is an important part of our design, will be beneficial to all other components. Other improvements include sharing the idle task between all servers, more efficient dummy events (allowing larger values and merging smaller dummy events), and adapting the scheduler to make use of μC/OS-II' 256 task extension.

Different scheduling algorithms. We currently support FPDS and FPPS, but other scheduling mechanisms could also be used for both global and local scheduling. For example, EDF has already been sucessfully implemented on top of our design in [44].

Port RELTEQ to and evaluate on other operating systems. It would be interesting to port RELTEQ to other real-time operating systems, such as VxWorks, and evaluate if and how much it can improve on overhead and reduction of memory footprint over the existing mechanisms used in those kernels. In our design, 40 lines of code were changed in the μC/OS-II kernel, and the code of some event handlers depend on OS-specific functions and structures (e.g. to acquire a semaphore), which would have to be adapted to the target OS. These kernel changes and event handlers would have to be implemented in another operating system. Furthermore, the design could be modified to become more

general, e.g. by allowing more than 8 servers and 8 tasks per server, and first steps have already been taken in this direction in [15].

One shot timer. Currently, we use μC/OS-II' periodic timer to drive the RELTEQ tick handler. We expect to be able to reduce the overhead of event handling event further by using one-shot timers. Specfically, the High Precision Event Timers (HPET) [21] promises to be a suitable timer, as the expiration times can be set relative to the previous interrupt, for which we can exploit RELTEQ's relative time model.

n-level HSF. We currently implement a two-level HSF. This can be extended further to support multiple levels. As we minimize the inteference from handling events of inactive servers, we expect this approach to be scalable to multiple levels as well.

Synchronization protocols. We would like to extend the design with support for additional synchronization protocols. Currently, resources can only be shared by using (H-)FPDS as a guard to critical sections. Various protocols have already been succesfully implemented on top of our design in [34, 44]. It might also be interesting to extend our H-FPDS implementation to support the local and hybrid H-FPDS variants, and to allow the selection of a H-FPDS mode at preemption point level.

References

1. R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, Resource kernels: A resource-centric approach to real-time and multimedia systems, in In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, pp. 150-164, 1998.

2. C. W. Mercer, S. Savage, and H. Tokuda, Processor capacity reserves for multimedia operating systems, tech. rep., Pittsburgh, PA, USA, 1993.

3. CANTATA Project. http://www.win.tue.nl/san/projects/cantata/ and http://www.hitech-projects.com/euprojects/cantata/

4. G. C. Buttazzo, Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series). Santa Clara, CA, USA: Springer-Verlag TELOS, 2004.

5. Liu, J. W. S. Real-Time Systems Prentice Hall, 2000

6. Lethbridge, T. & Laganiere, R. Object-Oriented Software Engineering: Practical Software Development using UML and Java McGraw-Hill, Inc., 2002

7. C. O. Pérez, M. Rutten, L. Steffens, J. v. Eijndhoven, and P. Stravers. Resource Reservations in shared-memory multiprocessor socs, volume 3, chapter 5, pages 109-137. Philips Research, 2005. ISBN 1-4020-3453-9

8. C. Mercer, R. Rajkumar, and J. Zelenka, Temporal protection in realtime operating systems, in Proceedings 11th IEEE Workshop on Real-Time Operating Systems and Software. RTOSS '94, Seattle, WA, USA, 18-19 May 1994 (IEEE, ed.), (pub-IEEE:adr), pp. 79-83, IEEE Computer Society Press, 1994..

9. M. M. H. P. van den Heuvel, Dynamic resource allocation in multimedia applications, Master's thesis, Eindhoven University of Technology, July 2009.

10. M. Bergsma, Extending RTAI/Linux with FPDS, Master's thesis, Eindhoven University of Technology, 2009.

11. M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, Towards hierarchical scheduling on top of vxworks, in Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08), pp. 63-72, July 2008.

12. Resource Reservation in Real-Time Operating Systems – A joint industrial and academic position, Steffens, Fohler, Lipari, Buttazzo, et al.

13. R. Rajkumar, Synchronization in Real-Time Systems: A Priority Inheritance Approach. Norwell, MA, USA: Kluwer Academic Publishers, 1991.

14. A scheme for scheduling hard real-time applications in open system environment, Z. Deng, J. Liu J.Sun, In proceedings of the 9th Euromicro Workshop on Real-Time Systems (EURO-RTS'97), pp. 191 - 199, 1997.

15. Mike Holenderski, Wim Cools, Reinder J. Bril and Johan J. Lukkien, Extending an Open-Source Real-Time Operating System with Servers and Hierarchical Scheduling, Technical Report, Eindhoven University of Technology, July 2010.

16. R. Davis and A. Burns, Hierarchical xed priority pre-emptive scheduling, in In Proceedings of the 26 th IEEE International Real-Time Systems Symposium (RTSS 05), pp. 389-398, 2005

17. Saewong, S. & Rajkumar, R. Hierarchical Reservation Support in Resource Kernels Real Time Systems Symposium (RTSS), 2001

18. M. Behnam, T. Nolte, M. Asberg, and R. J. Bril, Overrun and skipping in hierarchically scheduled real-time systems, in RTCSA, pp. 519-526, IEEE Computer Society, 2009

19. Klein et al, 1993. M. H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, A practitioner's handbook for real-time analysis. Kluwer Academic Publishers, Norwell, M.A. USA, 1993

20. D. Tsafrir, Y. Etsion, and D. G. Feitelson, Abstract general-purpose timing: The failure of periodic timers, Technical Report, The Hebrew University, 2005.

21. Intel. IA-PC HPET (High Precision Event Timers) Specification, 2004.

22. A. Carlini, G. C. Buttazzo. An efficient time representation for real-time embedded systems. In SAC '03: Proceedings of the 2003 ACM symposium on Applied computing, pp. 705–712. ACM, New York, NY, USA, 2003.

23. G. Buttazzo and P. Gai, Efficient EDF implementation for small embedded systems, in Proc. of the 2nd Int. Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2006), (Dresden, Germany), July 2006.

24. RTLinux. Rtlinux version 3.2. 2007. Online: http://www.rtlinuxfree.com.

25. SHaRK. Soft hard real-time kernel. 2006. Online: http://shark.sssup.it.

26. J. J. Labrosse. MicroC/OS-II: The Real-Time Kernel. CMP Books, 1998.

27. Shin, I. & Lee, I. Periodic Resource Model for Compositional Real-Time Guarantees Real-Time Systems Symposium (RTSS), 2003, pp. 2-13

28. Oikawa, S. & Rajkumar, R. Portable RK: a portable resource kernel for guaranteed and enforced timing behavior Real-Time Technology and Applications Symposium (RTAS), 1999, pp. 111-120

29. OpenRISC 1000 simulator. Online: http://opencores.org/openrisc,or1ksim

30. Holenderski, M.; Bril, R. J. & Lukkien, J. J. Using Fixed Priority Scheduling with Deferred Preemption to Exploit Fluctuating Network Bandwidth Work in Progress session of the Euromicro Conference on Real-Time Systems (ECRTS), 2008

31. Holenderski, M.; van den Heuvel, M. M. H. P.; Bril, R. J. & Lukkien, J. J. Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2010

32. M. van den Heuvel, M. Holenderski, W. Cools, R. Bril, and J. Lukkien, Virtual Timers in Hierarchical Real-time Systems Work in Progress session of the Real-time Systems Symposium (RTSS), 2009

33. M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien,. Multiplexing Real-time Timed Events Work in Progress session of the International Conference on Emerging Techonologies and Factory Automation (ETFA), 2009

34. M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien and M. Behnam, Extending a HSF-enabled Open-Source Real-Time Operating System with Resource Sharing, 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium, July 2010

35. μC/OS-II v2.86 port for OpenRISC. Online: http://www.win.tue.nl/~mholende/ucos/ucos_ii_2.86_openrisc.zip

36. A Guide to the OpenRISC Port of MicroC/OS-ii. Online: http://www3.ntu.edu.sg/home5/p014082819/or1k_ucos_howto.pdf / http://www3.ntu.edu.sg/home5/p014082819/download.html

37. Lehoczky, J. P.; Sha, L. & Strosnider, J. K. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments Real-Time Systems Symposium (RTSS), 1987, 261-270

38. R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in Proc. RTSS, Dec. 2006, pp. 257–270.

39. M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in Proc. EMSOFT, Oct. 2007, pp. 279–288.

40. Bril, R. J.; Lukkien, J. J. & Verhaegh, W. F. J. Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited – with extensions for ECRTS'07 – Technische Universiteit Eindhoven, 2007

41. A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," in Advances in Real-Time Systems, S. Son, Ed.Prentice-Hall, pp. 225–248, 1994.

42. Burns, A.; Nicholson, M.; Tindell, K. & Zhang, N. Allocating and Scheduling Hard Real-Time Tasks on a Parallel Processing Platform University of York, UK, 1994

43. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference (Online: http://developer.intel.com/design/processor/manuals/253667.pdf)

44. M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien, Protocol-Transparent Resource Sharing in Hierarchically Scheduled Real-Time Systems, 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), (to appear), Bilbao, Spain, September 2010

45. A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, Supporting time-sensitive applications on a commodity os, SIGOPS Oper. Syst. Rev., vol. 36, no. SI, pp. 165-180, 2002.

46. S. Kato, R. Rajkumar, and Y. Ishikawa. "A Loadable Real-Time Scheduler Framework for Multicore Platforms", In submission, 2010. Available online (http://www.contrib.andrew.cmu.edu/~shinpei/papers/rtcsa10.pdf)

47. D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino. An EDF scheduling class for the Linux kernel. In Proc. of the Real-Time Linux Workshop, 2009

48. Palopoli, L.; Cucinotta, T.; Marzario, L. & Lipari, G. AQuoSA—adaptive quality of service architecture Software – Practice and Experience, John Wiley & Sons, Inc., 2009, 39, 1-31

49. Eswaran, A.; Rowe, A. & Rajkumar, R. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks Real-Time Systems Symposium (RTSS), 2005, 256-265

50. Evidence. ERIKA Enterprise: Open Source RTOS for single- and multi-core applications. Online: http://www.evidence.eu.com

51. RTAI 3.6-cv - The RealTime Application Interface for Linux, 2009. Online: www.rtai.org.

52. Rajkumar, R.; Lee, C.; Lehoczky, J. P. & Siewiorek, D. P. Practical Solutions for QoS-Based Resource Allocation RTSS '98: Proc. IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998, 296

Appendices

Appendix A Programmer API (Periods, Reservations, FPDS)

The example code listings in the Appendix, show how to create a periodic task, server, set preemption points and use H-FPDS. To use the extensions, make sure to set OS_TEV_EN, OS_TEV_FPDS_EN and OS_TEV_SERVER_EN to 1 in OS_CFG.H, to enable RELTEQ & periodic tasks, FPDS, and reservations, respectively. OS_TEV_EN must be enabled to use any of the other extensions, and H-FPDS is enabled when all flags are set to 1.

Periodic Task.

```
void Task() {
  INT16U overruns = 0;
  OSTaskPeriodWait();
  for ( ; ; ) {
    /* .. task body here .. */
        overruns = OSTaskPeriodWait();
  }
}

int main() {
  /* .. create a task with priority TASK_PRIO here .. */
  OSTaskSetPeriodEx(TASK_PRIO, TASK_PERIOD, TASK_PHASE);
  /* .. start OS here .. */
}
```

Create server. Create a server with priority SERVER_PRIO, and given capacity and period. Available server types are OS_TEV_SERVER_TYPE_PERIODIC, OS_TEV_SERVER_TYPE_POLLING, and OS_TEV_SERVER_TYPE_DEFERRED. The mapping of a task τ_j to server σ_i, is implicit by creating a task in the task priority range of the server. OS_SERVER_PRIO(i, j) is used to map a new task with priority j to the server with priority i, where j is the priority of the task inside server i (i.e. relative to the other tasks in this server).

```
void Task(void *TaskArguments) {
  /* .. task code here .. */
}

int main() {
  OSServerCreate(SERVER_PRIO, SERVER_CAPACITY, SERVER_PERIOD,
        OS_TEV_SERVER_TYPE_PERIODIC);
  OSTaskCreate(Task, (void*)TaskArguments, &Stack,
        OS_SERVER_PRIO( SERVER_PRIO, TASK_PRIO ));
  /* .. start OS here .. */
}
```

FPDS.

```
void Task(void *TaskArguments) {
  OSTaskPeriodWait();
  for ( ; ; ) {
    /* .. subjob body here .. */
    OSTaskYield();
    /* .. subjob body here .. */
    if ( OSTaskShouldYield() ) {
          OSTaskYield();
    }
    /* .. subjob body here .. */
    OSTaskPeriodWait();
  }
}

int main() {
  /* .. create a task with priority TASK_PRIO here .. */
  OSTaskSetFPDS(TASK_PRIO);
  /* .. start OS here .. */
}
```

H-FPDS. We can use the overrun mechanism by setting OSServerSetFPDS, with the option PAYBACK_ENABLED (or PAYBACK_DISABLED), and specifying the overrun budget OVERRUN_BUDGET. Skipping is used by calling OSTaskYieldConditional(WCET), where WCET is the worst case execution time of the next subjob.

```
void Task(void *TaskArguments) {
  OSTaskPeriodWait();
  for ( ; ; ) {
    /* .. subjob body here .. */
    OSTaskYield();
    /* .. subjob body here .. */
    OSTaskYieldConditional(WCET);
    /* .. subjob body with worst-case execution time WCET here .. */
   OSTaskPeriodWait();
  }
}

int main() {
  /* .. create a server with SERVER_PRIO, containing a task with */
  /* priority TASK_PRIO here .. */
  OSTaskSetFPDS(TASK_PRIO);
  OSServerSetFPDS(SERVER_PRIO, PAYBACK_ENABLED, OVERRUN_BUDGET);
  /* .. start OS here .. */
}
```

Appendix B Component files list

μC/OS-II Source Listing:

Module	Source File
Core	OS_CORE.C
Time	OS_TIME.C
Semaphore	OS_SEM.C
Mutex	OS_MUTEX.C
Mailbox	OS_MBOX.C
Queue	OS_Q.C
Memory	OS_MEM.C
User Configuration	OS_CFG.H, INCLUDES.H
μC/OS-II Header File	UCOS_II.H
CPU C (Port)	OS_CPU_C.C
CPU Assembly (Port)	OS_CPU_A.ASM
CPU Header Files	OS_CPU.H

Extensions in this report:

Module	Source File
(H-)FPDS	OS_TEV_F.C
RELTEQ	OS_TEV_Q.C
Reservations	OS_TEV_R.C
Periodic Tasks	OS_TEV_P.C
Defines/Initialization	OS_TEV.H/OS_TEV.C

Binary:

μC/OS-II	39220
RELTEQ & Periodic Tasks	5792
Reservations	5352
FPDS	588
H-FPDS	348

Table 15: Binary sizes (in bytes) of extension components in compiled OpenRISC 1000 image.

Appendix C Method and function name mapping

For reasons of legibility, the naming convention of the methods used in this report is different from that used in the source code. In the source code, most methods start with the prefix *OSTimedEvent* to make sure their names do not overlap with methods of other modules. Furthermore, we try to follow coding conventions such as using CAPITAL letters for macros and compiler definitions. Finally, in some cases shortened names to keep the code listings more compact, or used longer names to prevent using unintroduced abbreviations. For providing a clear overview of the architecture and interface, we left out specific implementation details such as the coding conventions and whether a method is indeed implemented by a C-function or a macro definition. The following table provides a mapping of the interfaces named in this report, and their exact function names in the code.

Report	Source Code
ActivateLocalEventQueues	ACTIVATE_LOCAL_QUEUES
FirstNonVirtualEventTime	OSTimedEventGetFirstRealEvent
QueueInsert	OSTimedEventQueueInsert
QueueRemove	OSTimedEventQueueRemove
ServerSwitchIn	OSServerCtxSwitchIn
ServerSwitchOut	OSServerCtxSwitchOut
Multiplex	OSTimedEventUpdateFirstPointer
Demultiplex	OSTimedEventResetTimer
GetEventQueue	GET_RELTEQ
ServerContextSwitch	OSServerCtxSwitch
ResetCounterQueue	OSTimedEventStopwatchQueueReset
SyncEventQueues	OSTimedEventSyncEventQueues
IncreaseCounterQueue	OSTimedEventStopwatchIncrease
ServerCreate	OSServerCreate
ServerNameSet	OSServerNameSet
ServerMapTask	OS_SERVER_PRIO
ServerBudgetLeft	OS_SERVER_BUDGET_LEFT
GlobalScheduler	OSServerSchedule
LocalScheduler	OSServerLocalSchedule
SetFPDS	OSTaskSetFPDS
SetFPPS	OSTaskSetFPPS
PreemptionPoint	OSTaskYield
TickHandler	OSTimedEventTimeTick
QueuePop	OSTimedEventQueuePop
EventDelete	OSTimedEventFree
SetHFPDS	OSServerSetFPDS
ConditionalPreemptionPoint	OSTaskYieldConditional
WaitForNextPeriod	OSTaskPeriodWait
TaskSetPeriodic	OSTaskSetPeriodEx