Fabian Mauroner, BSc

Hardware Resource Management in FPGA based Multi-Core MCU Architectures

MASTER'S THESIS

to achieve the university degree of
Master of Science
Master's degree program: Telematics

submitted to

Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat.
Marcel Carsten Baunach

Institute for Technical Informatics

Graz, May 2015

Kurzfassung

In den letzten Jahren wurde von vielen Anwendungen immer mehr Rechenleistung verlangt. Auch im Bereich der eingebetteten Systeme spiegelt sich dieser Trend wieder. Eingebettete Systeme besitzen ganz andere Anforderungen als Hochleistungscomputersysteme. So beschäftigt man sich bei eingebetteten Systemen unter anderem mit diversen Echtzeitanforderungen, die meist in Software bzw. von den eingesetzten Betriebssystemen zu gewährleisten sind. Der Scheduler hat die Aufgabe diese Anforderungen unter Berücksichtigung der Software-Spezifikation zu bewältigen. Es gibt zahlreiche Scheduler die diese Aufgabe für komplett unabhängige Tasks sehr gut ausführen. Möchten jedoch mehre Tasks eine Ressource gemeinsam verwenden, ist die Aufgabe um einiges schwieriger. Um sie trotzdem zu bewältigen, werden sogenannte Ressource Management Protokolle verwendet. Diese Protokolle helfen dem Scheduler dabei die Tasks in der korrekten Reihenfolge abarbeiten zu lassen. In heutigen Rechnern findet man Multicoreprozessoren, die auch in eingebetteten Systemen immer öfter Anwendung finden. Die einzelnen Prozessoren auf einem Chip teilen sich den Zugriff auf gemeinsam verwendete Ressourcen wie etwa Speicher oder Datenbusse. Die meisten Ressourcen können nicht von verschiedenen Tasks gleichzeitig verwendet werden und müssen vor gegenseitiger Verwendung geschützt werden. Viele Mikrocontrollerherrsteller verschieben dieses Problem in die Software und so wurden diverse Vorgehensweisen und Implementierungvorgaben entwickelt. Bei deren Umsetzung nimmt die interne Verwaltung jedoch eine hohe Rechenleistung ein bzw. werden die Ressourcen den Prozessoren statisch zugeordnet. Allerdings arbeiten viele Geräte in einem sehr dynamischen Umfeld und somit ist eine statische Zuordnung nicht immer geeignet. In verschiedenen Industriesektoren, vorallem im Automobilsektor oder WSAN-Anwendungen (Wireless Sensor/Actuator Network) werden häufig Mikrocontroller verwendet. Außer der geringen Rechenleistung verfügen Mikrocontroller jedoch über sehr viele Vorteile die für Echtzeitsysteme notwendig sind (keine cache-misses, geringe Stromaufnahme, günstig in der Anschaffung, etc.). Da Mikrocontroller in bestimmten Anwendungsbereichen sehr gut geeignet sind, die Softwarelösungen wegen der geringen Rechenleistung oft jedoch nur schwer umsetzbar sind, wird in dieser Arbeit mit Hilfe einer Hardwareerweiterung das Ressourcen Management für das dynamische Umfeld von Multicoreanwendungen realisiert.

Abstract

In the last couple of years, applications have been expanding their need for computing power. This rising computing power is also needed in embedded systems, which have other requirements than high performance computing systems. A real-time application has to maintain the real-time constraints and the scheduler's job is to dispatch the tasks in the correct order to keep the time constraints. In systems where the tasks run independently of each other, the scheduler does its job well. However, in cases where two or more tasks have to use the same resource, more work has to be done for maintaining the constrained rules. To cope with this problem, there are many resource manager policies. The resource manager is often included into the kernel of an operating system and helps the scheduler to schedule the tasks into the correct order.

In high performance computing systems, one can find architectures with multi-cores. Also in embedded systems, the trend goes from single-core to multi-core architectures. The cores within a single chip have access to peripherals, buses and other components, which are accessible from all cores at the same time. These so called resources can not, in most cases, be simultaneously accessed by different cores, and this must be prevented reliably. Some microcontroller producers push this problem to the software level, where some approaches to handle it are available. These solutions on software level either demand high computing power or static assignments that result in static software. In contrast, the real world has a highly dynamical behavior. Moreover, microcontrollers have many advantages for real-time systems (no cache miss, low power consumption, low cost, etc.) but their computing power is low. The automotive domain or WSAN (Wireless Sensor/Actuator Network) are examples that use microcontrollers in a dynamic environment. In fact, microcontrollers suit well for many applications, but the software solution will not work on most of them, because it needs too much computation power.

This work presents a way to handle resources dynamically in hardware for multi-core processor architectures, keeping computation requirements low and, at the same time, maintaining all real-time constraints.

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

....

date (signature)

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Ich möchte mich von ganzem Herzen bei Sabrina bedanken, die mir in den letzten Monaten tatkräftig zur Seite gestanden ist und ohne die diese Arbeit nie zustande gekommen wäre!

Im Weiteren möchte ich mich bei meinem Betreuer Herrn Professor Marcel Baunach für die Betreuung dieser Arbeit und die umfangreiche Unterstützung bedanken.

Ein großer Dank geht auch an Renata Gomes für das Korrekturlesen der Arbeit.

Zuletzt möchte ich mich noch bei meiner Familie und meinen Freunden bedanken die mich in den letzten Jahren unterstützt haben und immer für mich da waren.

DANKE!

Contents

1. Introduction

Multi-core development has been an important research domain in the last couple of years. More cores on a die make it possible to reduce the power consumption while even increasing the computational power. In addition, the multi-core environment allows one to bring more applications into a one-chip solution, the so called System on Chip (SoC). Different domains are interested in multi-core SoC solutions, like wearable devices, WSAN, or the automotive sector. It does not only reduce the size of the final device but also the production costs.
Most research in the area of multi-core systems was done for high performance systems, like workstations. In almost every new computer a multi-core CPU is integrated. A workstation is designed for control structures and especially for high performance computation. Therefore, the used CPU in a workstation is a general-purpose processor. However, an embedded system has, in most cases, totally different requirements as a workstation. Hence, embedded systems use a MCU. A MCU is a computation unit that is designed primarily for control applications, and the embedded system is often used as a control unit. Control units are used in different safety critical applications. Therefore, most embedded systems are also real-time systems. A real-time system must guarantee that an incoming event is computed within a defined time, otherwise severe consequences might follow. The real-time scheduler, which is well investigated, has to maintain time constraints.

1.1. Terminology

Scheduling

For a single-core real-time system a lot of different scheduling approaches with all their benefits and disadvantages exist. The scheduler has the job to schedule all tasks according to their priorities to meet all time constraints. The priority of a task shows how important a task is and there are many different approaches to define this priority. For real-time systems there are two common approaches. One approach sets a static priority (SP) to every task, while the Earliest Deadline First (EDF) approach sets a dynamic priority to every task. The dynamic priority depends on the absolute deadline of a task. If the deadline of a task A is earlier than the deadline of task B, task A gets a higher priority than task B. On a multi-core environment these two approaches can be used in three different manners. One possibility is to use the scheduler globally (G-EDF or G-SP), where all tasks could run on any core. The scheduler decides which task is moved to which core. Another approach is partition scheduling (P-EDF or P-SP). All tasks are assigned statically to a core, where an independent scheduler runs and decides which task should run next. Finally, the two approaches could be mixed into so-called clusters (C-EDF or C-SP). The cores are split into clusters and the tasks are assigned to a cluster. This means, that the task could run at all cores that are in its assigned cluster. As in partitioned scheduling, on each cluster runs an independent scheduler. In the partial and clustered scheduling, the different schedulers on each core do not have to be of the same type. Therefore, the systems on every core or cluster are totally independent from each other.

Task

A task can be periodic or sporadic. In the case of a periodic task, the task's release time is on every regular interval time, while the sporadic task release time is irregular. A task can be in different states. If the task runs on the core, it is in the *running* state. In the *ready* state are all tasks that are ready to run, but can not yet run, e.g. when it has been preempted by another more important task. A the task can also be in waiting state after suspending itself. Suspension occurs when a task requests a resource that is not free, calls a sleep function, or waits for an event.

Resources

Resources are components which are on a computer system. There are many different kinds of resources with different characteristics. Some resources have to be used with an exclusive access, to avoid functional problems or race conditions. Some resources, like the CPU, could be used in a preemptive way. Many resources must be initialized and closed in a correct order and therefore, a preemption could bring the resource to an invalid state. To ensure that the resource is used exclusively, many approaches exist. One well-known approach are *semaphores*. Two operations are used to allocate and deallocate a resource. A task can *request* for a resource and, if it succeeds, the resource is *allocated* to the task that continues execution using the resource. Otherwise, if another task holds the resource, the request will be refused and the task will have to wait until the resource manager hands over the resource. Sha et al. [23] defines the term *blocking*. A task is called blocked if it makes a resource request and has to wait for a lower prioritized task that holds the requested resource. If the task waits for a higher priority task, because of a resource request, the task is not called blocked. A task is only called blocked if the priority inversion problem occurs. Section 2.1 shows some approaches to minimize the blocking time. On a multi-core environment, the term blocking is extended to *global blocking*, by Rajkumar et al. [22]. A task is called globally blocked if it has to wait for its requested resource on every prioritized task. Section 2.2 shows approaches to reduce this kind of waiting on a multi-core environment. On a multi-core environment, the resources are distinguished into local and global resources. The local resources are only on one core and are handled with a single-core resource management protocol. The global resources are shared over more cores and are handled by a multi-core resource management protocol. The section between the allocation and release of the resource is called *critical section* and is distinct between local (lcs) and global (gcs) critical sections. Furthermore, a resource could be used by a task as a *short-term* or *long-term* resource allocation [2]. If the task suspends itself in a critical section, e.g. sleeping, the task uses the resource in a long-term way. Otherwise, it is called short-term allocation. A *long-time* resource allocation is a *long resource* [9], which is a allocated resource with a long critical section. If another resource allocation is done inside a critical section, it is called *nested* resource allocation.

1.2. Problem

In systems where tasks are independent of each other, the scheduler does its work very well. However, if the tasks have to be synchronized or a resource is concurrently accessed, a synchronization mechanism is needed. Many methods, such as semaphores, locks, monitors and Ada rendezvous [23], can be used for synchronization or for resource allocation. The synchronization mechanisms causes a dependency between two or more tasks. This leads to the well-known priority inversion problem [23, 5].

Priority inversion

The priority inversion problem occurs on a preemptive and prioritized task system [4]. The highest prioritized task H preempts the lower prioritized running task L if it switches from suspended state to ready state. This is the desired run. If the task H wants to allocate the same resource that task L holds, the desired run is not possible anymore. Therefore, the task H is blocked until the task L releases the requested resource. This blocking is also called *priority inversion*, because the lower prioritized task L runs instead of the higher prioritized task H [23]. At bounded priority inversion, the maximum delay time is the maximal time of task L's critical section. Figure 1.1 shows an example of a bounded priority inversion. Further, [2] defines the unbounded priority inversion problem, where an additional task M is involved. The task M

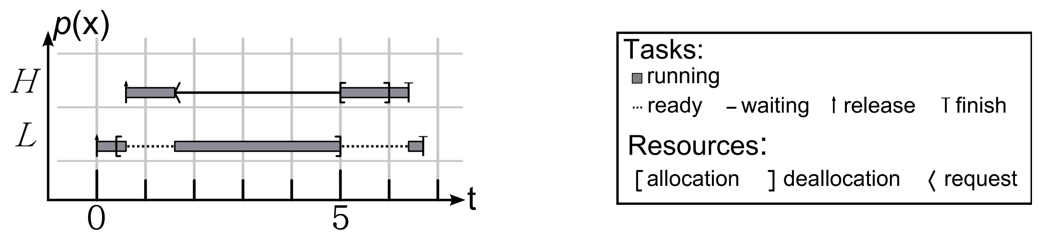


Figure 1.1. – Bounded priority inversion.

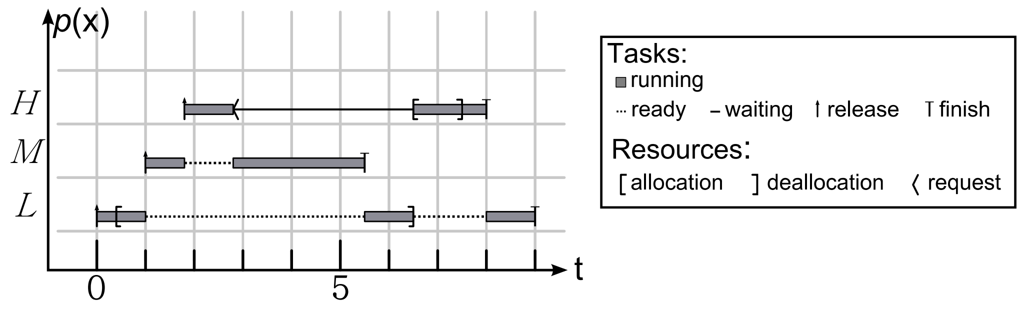


Figure 1.2. – Unbounded priority inversion.

preempts the task L that holds the resource that task H wants. Therefore, task H has to wait until task M has finished the execution and the task L releases the resource. The blocking time of task H increases about the execution time of task M. Figure 1.2 shows an example of an unbounded priority inversion. The two examples showed that the task H is resumed at a later point. However, in the case of an unbounded priority inversion the maximum delay of the task H is the critical section execution time and additionally the execution time of any tasks that can preempt task L, which makes the delay unpredictable. This leads to unpredictable behavior, may lead to deadline violations and even to deadlocks, which is not acceptable in real-time systems. The priority inversion problem has to be handled with so-called *resource protocols* that reduce the blocking times, and guarantee that no unbounded priority inversion happens.

Long-term and Short-term resource allocation

In a whole system, many resources are used by different tasks at the same time. Therefore, some tasks would be blocked. The duration of a blocking period depends on the length of the critical section and the execution time of higher prioritized tasks. There exist some synchronization protocols that minimize the blocking time. The time of the critical section is the execution time inside a task's critical section, but only for a short-term allocation, when the task does not suspend itself while running inside its critical section. If the task suspend itself, we speak about a long-term allocation [2]. The long-term allocation brings the problem that the blocking time increases about the time where the task is suspended.
This problem can be prevented if long-term allocations are not allowed, but this is not always possible. In highly dynamic environments, it is often necessary to hold resources for long-terms. A common example is shown in Figure 1.3. Suppose that a log task uses the SPI bus to save logging data on an SD-card, on which a file system runs. The motor controller uses the SPI bus sporadically after receiving an event from the environment. The user task communicates among the UART port with a user and uses the SPI bus to operate on the SD-Card. There exist a lot of different approaches to design this behavior. One solution is to divide the bus into

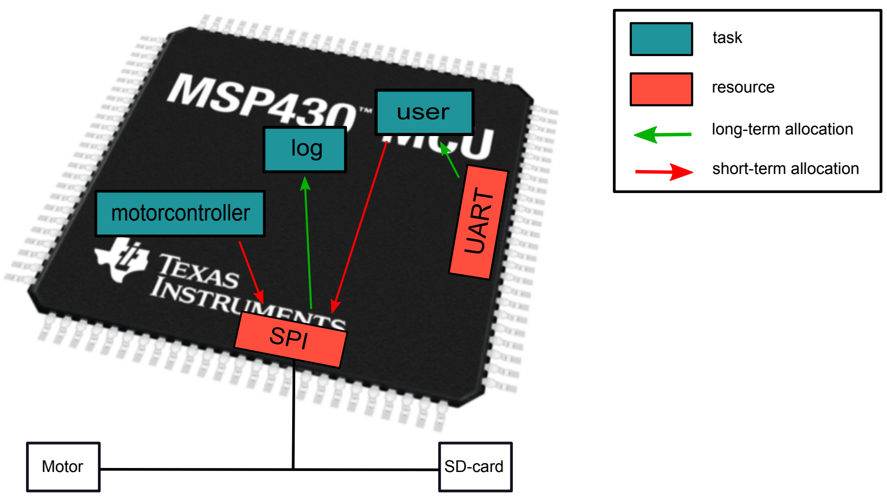


Figure 1.3. – Short- and long-term resource allocation example.

time-divisions. However, this increases the power consumption, because drivers have to run the header and tail on every time-slot. Furthermore, it is hard to decide how long a slot should be and how many slots should be assigned to a task. On a dynamic environment, a time-slot could be used only rarely, like the motor controller or the user access to the SD-card. Therefore, some slots would be left free. Thus, it is necessary to manage this in a more dynamic way.

Single-Core Protocols on a Multi-Core Environment

Single-core protocols are well researched and are used in many commercial and non-commercial operating systems. Therefore, the idea to use the same protocols on a multi-core environment is straightforward. Figure 1.4 shows the usage of the single-core resource protocol Priority Ceiling Protocol (PCP) on a multi-core environment with two cores. On one core c_0 runs task H and task L; on the second core c_1 runs task M. The task L is released at time 0 and before the task M is released, the resource is allocated by the task L. The task M requests the resource, but it is already assigned to the task L and therefore the task M suspends itself. The ceiling priority of the resource is equal to task M's priority, because only task M and task L use it. At time 2, the task H is released. Task H has higher priority than the actual highest ceiling priority on the system and therefore, task L is preempted. At this point the problem of a single-core resource protocol occurs. The waiting time of the task M, until the allocation of the resource, depends on the critical section time of the lower prioritized task as well as on all non-critical section times of higher prioritized tasks on *other* cores outside a critical section. The aim of many resource management protocols is to bound the blocking time to a function of critical sections and not on non-critical codes [22]. Therefore, the possibility to test the system for schedulability at development time is very pessimistic.

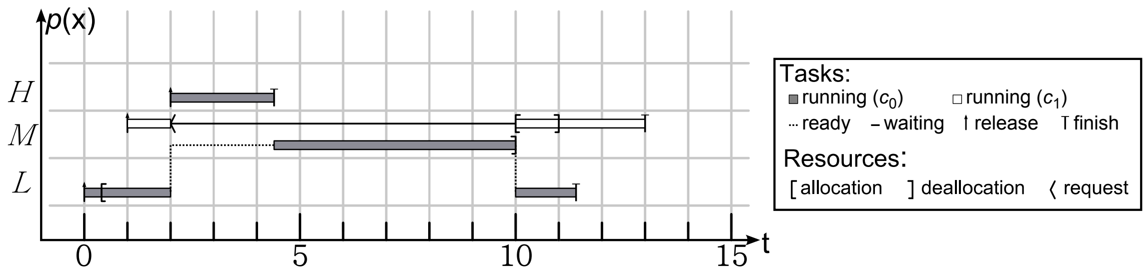


Figure 1.4. – PCP on a multi-core environment.

1.3. Outline

Chapter 2 presents some related works on resource management protocols and hardware resource management peripherals.
In the next Chapter 3, the fundamentals for this work are introduced. The specification for the whole system is specified. Besides, the base resource management protocol is described in detail. The chapter concludes with the used technologies for the implementation of the proposed approach.
Chapter 4 is the main part of this work and presents a new resource management protocol, which is an extension of the base protocol presented in Chapter 3. The chapter concludes with the implementation details of the FPGA and the software.
Chapter 5 contains the evaluation of the proposed resource management protocol. Firstly, the used test-bed is presented and afterwards, the Hardware Resource Manager (HWRM) is tested with well-directed stimulus. The chapter concludes with a case study of a general application.
Chapter 6 concludes the work with some potential future works and a summary of this work.

2. Related Work

In the past, the need of embedded real-time systems has grown. Therefore, multitasking systems were invented. If more tasks use shared resources, they may interfere on each other and the priority inversion problem may occur. To reduce this priority inversion problem, some synchronization protocols were created. In the following section, some protocols for a single-core environment are presented. Section 2.2 shows some protocols that could be used on a multi-core environment because they eliminate the single-core protocol usage problem on a multi-core environment. In the last section of this chapter, some solutions for a hardware resource management are shown.

2.1. Resource Management on Single-Core Architectures

Sha et al. [23] invented the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP). Both protocols are developed for a fixed priority scheduling [19]. The idea of PIP is that the priority of the task temporally rises to a higher priority than its base priority. If a task L is in a critical section and a higher prioritized task H is blocked because of a resource request, the task L inherits the priority of the task H. After releasing the resource, the temporally risen priority falls back to its base priority. Consequently, the task H is blocked for at most the duration of the critical section of task L [23]. Now it is possible to use the calculated maximal blocking time to get a schedulability test at compile time. The PIP has two problems that have to be handled. Firstly, PIP does not prevent deadlocks. Suppose a task 1 has the resource 1 and the task 2 has the resource 2. At a later point, task 1 makes a request for resource 2 and task 2 makes a request for resource 1. This causes a deadlock, as both tasks will wait forever for the requested resources and never release the resource they already own. Secondly, the PIP could cause a chain of blocking. This means that the higher prioritized task has to wait for more than one critical section of two or more lower-prioritized tasks.
These two problems are eliminated by using the PCP [23]. PCP defines a ceiling priority for every used resource in the system. The ceiling priority of a resource is the highest task priority which may use the resource. A task H can only enter into a critical section if its priority is higher than the highest ceiling priority of resources already assigned to some tasks. Otherwise, the task will be blocked. The task that does not cause a priority inversion runs with its base priority until a priority inversion occurs. Then, the task inherits the highest task priority which

is blocked at a resource request. This is the basic idea behind PCP, a more detailed description can be found in [23].
The Stack Resource Policy (SRP) from Baker is an extension of the PCP. This protocol can be used with read-writer locks, multiunit resources, a unique system wide stack and of course with binary semaphore, like PCP does. Furthermore, SRP could also be used with an EDF-scheduling. This protocol is not essential to understand this work, but for more details, I forward to [1].
In all presented protocols the critical section cannot be released by another more important task. This leads to the problem that the blocking time of a critical section could be very long or, much worse, the task suspends itself while it is in a critical section, e.g. timeout or waiting for an event. Hence, the schedulability analysis has to keep this in mind. Baunach [5, 4, 3, 2] invented a quasi-preemptive resource sharing, the so called dynamic hinting. The dynamic hinting approach could be combined with PIP or PCP. In this approach, the protocol works like the base protocol, but the task could be notified with a hint from its assigned resources. If the request of a task is refused, the blocking task, which causes a priority inversion receives a hint. The hint contains information as the requester's priority or the maximal waiting time for the resource request. Furthermore, in the case of PIP, a flag shows if a deadlock occurs. Thus, the receiver can decide if collaborating or not. The hint can be received by calling a function explicitly, an early wake up or by a task-defined hint handler. With this approach, the decision of collaborating is totally defined in the context of the task that uses the resource. Therefore, the long-time or long-term allocation is split dynamically by the task itself.
SmartOS, an operating system developed at the University of Würzburg, includes the dynamic hinting approach [8, 2]. It has been developed especially for time-critical embedded systems.

2.2. Resource Management in Multi-Core Architectures

On a multi-core environment, the resource management has to control the access to a resource by more than one core simultaneously. The research has gone in various directions. One approach is to add a software layer [11, 16, 21, 13], that manages and forwards the entire requests from the virtual processor on the top, to the physical multi-core environment at the bottom. This leads to a huge computation power that does not exist on a MCU. This virtualization approach uses a global scheduling proceeding. There is a global point in the system, where the scheduling and therefore, the resource management is handled. The global scheduler decides which task runs on which core. The opposite of a global scheduler is partial scheduling. In this approach, the scheduler is assigned to a core and therefore, every core runs independently from each other. Lastly, the hybrid scheduling is a mix of a global and a partial scheduling proceeding. A scheduler manages more than one fixed assigned core but the schedulers are independent from each other. In this thesis, I will focus only on the partial scheduling approach.

Rajkumar, Sha, and Lehoczky [22] extended the PCP to a multi-core version called Multiprocessor Priority Ceiling Protocol (MPCP). This protocol works very similar to the single-core version, with one major difference. In the single-core version, the tasks inherit the priority from a higher prioritized task that makes a request for a common resource. Then, the task runs temporally with the inherited priority. In the multi-core version, the tasks uses the highest priority in the system and the ceiling priority to calculate its temporary priority. This is also called priority boosting. I do not go into details here, but more precise information can be found in Chapter 3 and in [22].
The SRP multi-core version is called Multiprocessor Stack Resource Policy [14], and it has the same properties as SRP in the single-core version. A task that requests a resource performs a busy-waiting instead of self-suspension. This prevents other tasks from executing their code on the same core and therefore, is it only applicable for very short critical sections. Gai et al. [14] shows that Multiprocessor Stack Resource Policy (MSRP) performs even better than MPCP for short critical sections.
Block et al. [9] introduced the Flexible Multiprocessor Locking Protocol (FMLP) that could be used on a global and partial scheduling. This protocol performs a busy-waiting for short-time resources and a self-suspension for long-time resources for a rejected resource request. The programmer has to define which method should be used for each resource. The resources are grouped into different groups that are protected by a group lock. In a group, it is only possible to have all resources of short-time type or of long-time type. If a task is in a critical section, it becomes non-preemptible, because this protocol does not support priority inheritance. In many cases, it is hard to decide which type should be used for a resource.
$\mathcal{O}(m)$ Locking Protocol (OMLP) was invented by Brandenburg et al. [10]. The task has to hold a virtual local resource to get access to a global resource. The virtual local resource exists only once for each core. Every global resource has a FIFO queue and, on each core, there is a priority-based queue. The task on the head of the local queue that assigns the virtual local resource is also added to the global queue. Furthermore, its priority is boosted. Finally, the first task in the global queue holds the global resource. This protocol is asymptotically optimal, this means that in the worst case the blocking time for any task at any time has a constant upper bound.
The Multiprocessor Synchronization protocol for real-time Open Systems (MSOS) in [20] shows how resources could be shared on an independently developed real-time system. When the task gets a resource, its priority is risen immediately. Therefore, it can be preempted only from a higher prioritized task that is in a critical section, as in MPCP. MSOS defines that every resource has a global FIFO queue and a local priority queue in each core. Every time a task on a core requests a resource, it is added to the local queue and, on the global queue, the reference to the core is added. The head of the global queue gets the resource and consequently the task with the highest priority in the local queue gets the resource.

All of the presented protocols protect against deadlocks. Furthermore, they all reduce the time of a priority inversion to at most one critical section of lower prioritized tasks. However, the problem for long-term and long-time resources remains. These restrictions are going to be disposed in this work.

2.3. Hardware Resource Management

Freescale includes a Hardware Resource Manager (HWRM) into their MCP56xx and MPC57xx series [24]. The semaphores, named in theses context *lock gates*, are requested and released with a write operation to a register. To get a resource, the task has to write the number of the core into a register and then the core status register shows if the semaphore is assigned to the requested core. The resource is released when the task writes a special pattern to the same register.
The XGATE-family from Freescale proposes another technique [18]. Two instructions are used to set and to release the semaphores. After using the set semaphore opcode, the carry flag in the status register shows if the request was successful. Thus, the requester has to check the carry flag to know if the semaphore is hold by the core. This technique has the benefit that the code is thread safe, because the status register is save in an interrupt or in a context switch.
In [25, 17], Texas Instruments shows their used semaphore hardware module. There is one register for each resource that has to be used for requests and releases. The module includes three different modes to access semaphores. In the first mode, *direct* semaphore request, the request is done with a simple register reading. The read value shows if the core got the resource, or which core holds it. The second mode is the *indirect* request. The request is done with a write operation to the register. An interrupt notifies if the requested resource is granted. This could be immediately after the request or at a later point in time. The third mode is the *combined* mode, which is the combination of the two other modes. To request the semaphore, the application has to read out the register. If the semaphore is free, the return value shows this. Otherwise, the request goes into a queue and the core will be notified with an interrupt when the semaphore is granted.
None of this, in commercial used approaches, uses a resource management protocol in its HWRM. Therefore, the management protocol has to be implemented into software level to each core. This brings a huge communication overhead for information about the priorities of each task that the resource management protocol needs.
The dynamic hinting concept was also implemented into a multi-core environment. Baunach [7, 12] uses an interrupt to indicate a hint, in case a more important core wants the resource. The received hint is forwarded to the task that holds the resource and it decides to release the resource earlier or not. The used resource management protocol is PIP and this brings some problems with it, as for example deadlocks. Furthermore, the blocking time of a task could be

longer than one critical section of a lower prioritized task.
Hence, in this work I am going to combine the resource manager protocol for multi-core environments with the hardware resource manager to overcome the prior showed problems.

3. Background

In the first section the definition for the used system is specified. The next section introduces the Multiprocessor Priority Ceiling Protocol, the base resource protocol for the proposed Hinted Multiprocessor Priority Ceiling Protocol (HMPCP) in Chapter 4. The used platform for the FPGA implementation, the openMSP430, is introduced in Section 3.3. Finally, the last section is going to show the used operating system in which the concept was integrated.

3.1. System Specification

In this section, the properties of the used system are formalized. The system consists of a set of tasks T, cores C $(n := \#C)$, operating systems S and global resources R $(m := \#R)$.

Task

The tasks are scheduled independently on every core with a fixed priority (P-SP). Therefore, the number of operating systems S is equal to the number of cores C, $(\#S \overset{!}{=} \#C)$. The set of tasks T_i on a core $c_i \in C$ is defined as follows:

$$T_i \subset T \land \{T_i \cap T_j | \forall j = 0...n-1 \land j \neq i\} = \emptyset \tag{3.1}$$

A task $t_{ij} \in T_i$, where i is the core the task runs on and j is the task's priority, is preemptive with its defined WCET E_{ij}, relative deadline D_{ij}, and the blocking time B_{ij}. It could be sporadic or periodic, and if it is periodic, the period $\tilde{T}_{ij}$ is defined. Because of a static priority scheduling, the *base priority* of a task $P_{t_{ij}}$ is defined at compile time. The priority is ordered as follows:

$$P_{t_{ok}} < P_{t_{pl}} \text{ with } k < l \land o, p \in C \tag{3.2}$$

This means, as higher the priority number is as higher is their priority and that is applied for the whole system. The active priority $p(t_{ij}) \geq P_{t_{ij}}$ of a task t_{ij}, the priority that is used for scheduling by the scheduler, is dynamically modified by the resource manager. At start-up the active priority is equal to its base priority: $p(t_{ij}) = P_{t_{ij}}$

Resources

All resources $r \in R$ are shared across all cores in a non-preemptive way and are protected with semaphores. The resources can be assigned only exclusively to one task at a time thus, to one core. If a resource is assigned to a task t, the task becomes the task-resource owner

$$\sigma_r = \begin{cases} \emptyset & \text{if } r \text{ is not assigned to a task} \\ t \in T & \text{if } r \text{ is assigned to task } t \end{cases}. \tag{3.3}$$

The tasks t are statically assigned to a core c therefore, this leads to the core-resource owner

$$\Theta_r = \begin{cases} c_i \in C & \text{if } \sigma_r = t \land t \in T_i \\ \emptyset & \text{otherwise} \end{cases}. \tag{3.4}$$

Nested locks are allowed, therefore a task has the possibility to allocate more resources inside its critical section. The resources hold by a task t are summarized in:

$$R_t = \{r \in R | \sigma_r = t\} \tag{3.5}$$

If a tasks allocation is refused, the task is inserted into a waiting queue for the resource r. The highest priority of a task t that is waiting for the resource r is defined as:

$$w(r) := \max\{0, P_t \mid t \text{ waits for r}\} \tag{3.6}$$

3.2. Multiprocessor Priority Ceiling Protocol

This section introduces the multiprocessor version of PCP that was developed by Rajkumar et al. [22].

MPCP guarantees that the highest prioritized task that requests for a resource, waits at most for one critical section time of a lower prioritized task. Therefore, the maximum blocking time can be calculated and the schedulability analysis shows at development time if all deadlines could be maintained.

The protocol has to know all tasks that use a resource. This could be hardcoded at development time, but a better and more dynamic approach is to let task t register itself for every in-future-needed resource r at startup:

$$\rho_r = \{t \in T | t \text{ uses } r\} \tag{3.7}$$

With the knowledge of ρ_r the ceiling priority $c(r)$ of all resources r is calculated as:

$$c(r) := \begin{cases} 0 & \text{if } \rho_r = \emptyset \\ \max\{P_t \mid t \in \rho_r\} & \text{else} \end{cases} \tag{3.8}$$

The ceiling priority for every resource r is the maximal base priority of all tasks that use the resource r. The calculation of all ceiling priorities $c(r)$ results in the maximal ceiling priority

$$H = \max\{c(r) \mid r \in R\}. \tag{3.9}$$

The single-core version of PCP defines the current system ceiling priority

$$M = \max\{0, c(r) \mid r \in R \wedge \sigma_r \neq \emptyset\}. \tag{3.10}$$

In the original version from Rajkumar et al. [22], nested locks are not allowed. Therefore, for a successful resource request, the requested resource has to be free and the task priority has to be higher than the current system ceiling priority M. To allow nested locks, it needs an extension. To get a global resource and to allow nested locks, Equation (3.11) must hold. The resource r is assigned to the task t if firstly, the resource is free and if secondly, the base priority P_t is higher than the ceiling priority of all resources that are assigned to a task, excluding the resources assigned to the requester task t.

$$\sigma_r = \emptyset \wedge P_t > \max\{0, c(s) \mid s \in R \wedge \sigma_s \neq \emptyset \wedge \sigma_s \neq t\} \tag{3.11}$$

If Equation (3.11) cannot be maintained, the task suspends itself and is inserted itself into a priority waiting queue. The task is resumed if it is the head of the priority waiting queue and the condition is true.
To ensure that the priority inversion of a task is maximum one critical section of another task, the priority of the task has to be adjusted dynamically at run-time. The priority of the task, which causes a priority inversion, has to change. Rajkumar et al. [22], proposes to change the priority already at the beginning of the critical section. This reduces the management overhead, but could prevent some independent tasks from executing. A better solution is to adjust the priority only if it is needed. Therefore, the priority is changed when is changed as soon as a priority inversion occurs, i.e. when a higher prioritized task is inserted into the waiting queue. In the PCP the task that causes a priority inversion inherits the priority of the blocked task. On the other hand, in the multi-core version, this is not possible, because of the problem shown in Section 1.2. Thus, the critical section should not be preempted by other tasks that are running outside their critical section [22]. To keep to this, the active priority of task t is calculated

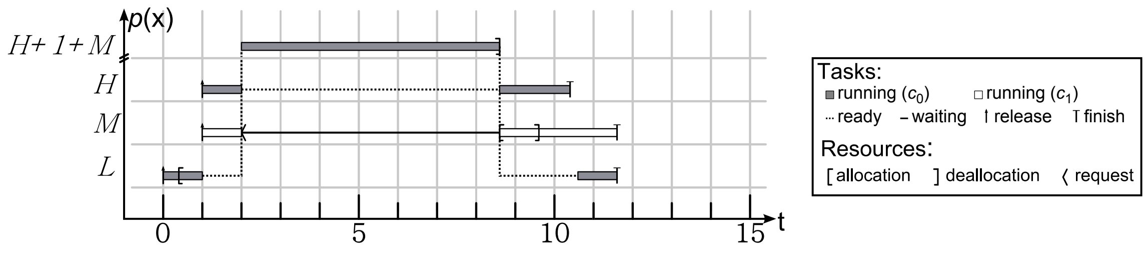


Figure 3.1. – Multiprocessor Priority Ceiling Protocol example.

according to Equation (3.12), which in the literature is called *priority boosting*.

$$p(t) := \begin{cases} H + 1 + c(r) & \text{if task } t \in T \text{ causes priority inversion on a resource } r \\ P_t & \text{otherwise} \end{cases} \tag{3.12}$$

Figure 3.1 demonstrates an example of MPCP. Consider that three tasks run on a system. Task M is assigned to core c_1 and task L and task H are assigned to the core c_0. Task H is the highest prioritized task and task L the one with the lowest priority in the system. Task L is released first and requests a resource. It immediately gets the resource because it is free. Afterwards, task M on the other core is resumed. Consequently, both tasks are running on their assigned core. At time 1, task H is released and preempts task L in its critical section. Until now, no priority inversion occurs. Thus, all priorities of the tasks are the same as their base priority. At time 2 task M requests for the resource that is assigned to task L. Task M is suspended and another task could run on the core. Now, a priority inversion occurs: the higher prioritized task M waits for the lower prioritized task L. Hence, the priority of task L is boosted and preempts task H. After releasing the resource, the boosted priority falls back to its base priority and the resource is overhanded to task M. Thus, the blocking time of task M is, in the worst-case, the WCET of task L's critical section.

3.3. openMSP430

Girard [15] has developed the openMSP430 project[1]. It is an instruction compatible open source implementation of the MSP430 MCU, which is a 16 Bit RISC architecture from Texas Instruments. It has only a few differences to the original, as the cycles of all operation are not exactly the same. The project includes a debugger and some peripherals, which are written in Verilog. In addition, it can be synthesized for a FPGA or for an ASIC. For the code generation the same compiler as for the original MSP430 can be used, such as GCC, a well-known open source compiler for the MSP430. Furthermore, the project adds the support for a multi-core environment. Every instantiated core communicates with only one debug interface. Every core

[1]`http://opencores.org/project,openmsp430`

has a program and memory interface that is connected to a memory. Further, each core has its own interface to communicate with the peripherals.

3.4. SmartOS

SmartOS [2, 8] is an operating system developed at the University of Würzburg. The aim of this operating system is time awareness and resource management, which is important for dynamic systems, as Wireless Sensor Networks (WSNs). SmartOS brings many new concepts, for example the timestamping of interrupt events [6]. Timestamps are very useful for a real-time system, because they bring a time sense into the application and could be used to guarantee the reaction of an interrupt within a defined deadline. The timestamp concept in SmartOS reduces the discretion time error of the timestamps in average to zero [6]. This leads to more precise timestamps.
Hardware interrupts are traditionally handled at a privileged mode in the kernel. This can slow down the whole system, because the kernel can not be preempted by a task. If an interrupt is handled for a low prioritized task, a higher prioritized task has to wait for the kernel exit. This is similar to a priority inversion. To eliminate this problem, the interrupts are forwarded to application level and handled there. Therefore, interrupts can be preempted by a higher prioritized task.
To synchronize tasks, SmartOS introduces events and mutexes. Moreover, SmartOS adds exception handling in a pure C written code. This makes the program more readable because the error handling is handled at one point. Another focus of SmartOS is the resource handling. The resources can be accessed with two simple methods: get and release. Moreover, the get function allows to wait a maximum time for the resource request. This enables the programmer to define hard real-time constraints that the operating system tries to manage. The return value shows if the time is over or the resource is assigned to the core. At compile time, the programmer could decide if he uses PIP or PCP as resource protocol. To reduce priority inversions, SmartOS adds the dynamic hinting approach.

Dynamic Hinting

Priority inversion is a major problem for real-time systems. Some resource protocols reduce the blocking time to a maximum of one critical section execution time of lower prioritized tasks. Traditionally, the critical section cannot be divided. Therefore, the time in the critical section could be very long for long-time and for long-term resource allocation. For this reason, the dynamic hinting approach was created. The non-preemptive resource is handled in a quasi-preemptive way. The task has the power to decide if it releases the resources earlier or not, based on the blocked task's priority and on its situation. This is important, because the task has to bring the resource back to the same status as before, when it resumes with the resource. SmartOS provides three methods, which allow the task to react to a hint. The easiest way is to query the hint at specific positions in the code. The explicit query does not work well for long-term allocated resources. In the time where the task is suspended, it makes more sense to give the resource to a more important task if it needs it. Hence, the kernel calls, e.g. `sleep`, `waitEvent`, `getResource`, could return earlier as the defined timeout. The return-value shows the occurrence of a hint and the task could query the hint information as in the first approach. There, the hint is handled by an explicit query or by a self-suspension call that resumes earlier. However, if the code runs for a long time without explicit query and no kernel calls, the hint is not received. Therefore, a third approach exists, where a handler manages the hint. This concept works very similarly to an interrupt, but the handler runs in the context of the task. The handler could be changed dynamically at runtime and could therefore react to a hint accordingly to the actual state of the task.

4. Hinted Multiprocessor Priority Ceiling Protocol

This chapter will show the extended specifications and operations of my proposed HMPCP that is based on the MPCP from Rajkumar et al. [22]. The first section extends the resource management protocol with timeouts, because the base protocol does not support timeout operations. In the second section, I combine the base protocol with dynamic hinting, developed by Baunach [5, 4, 3, 2]. The last three sections show the structure of the developed resource manager and the implementation details of the FPGA and the software extension.

4.1. Timeout

Timeouts are very useful features in an operating system and can be used to define hard real-time constraints. A task suspends itself for a defined maximum time if the resource allocation request was negative. Thus, the task does not prevent other tasks to run on the core while it waits for a resource. Furthermore, the maximum waiting time is limited. This means that a refused request cannot persist in a forever waiting if the resource is never released by the owner task.

The HMPCP also has to manage the timeout operation. If no priority inversion on a resource occurs anymore because of a timeout, the boosted priority has to be revoked to the base priority. Figure 4.1 shows an example of the priority fallback. The task L runs on core c_0 and the task H runs on core c_1. At time 0 task L is released and before task H is released, the resource is

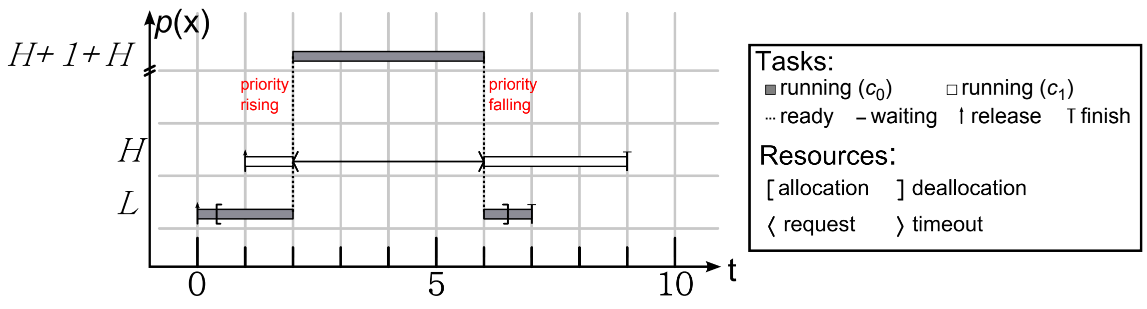


Figure 4.1. – Example of a priority rising, because of a priority inversion and a priority falling after the elimination of the priority inversion.

assigned to task L. Between time 1 and time 2 both tasks run continuously on different cores. At time 2 task H requests the resource owned by task L. Since the resource can not be allocated by task H, the priority of task L is increased to its boosted priority. Task H waits for a maximum time of four time slots, afterwards its request is expired. At this point, no priority inversion occurs anymore and the boosted priority is no longer needed. Task L priority falls back to its defined base priority.

4.2. Dynamic Hinting

Many resources are non-preemptive and some tasks could hold a resource for a long-term. In the case that the task suspends itself with an assigned resource, the blocking time of a waiting task could increase dramatically. Therefore, the resources should be released earlier if a priority inversion occurs. However, this has to be decided by the task. It is important that the resources are not released by the resource manager or by the operating system, than in this case the resources are handled in a preemptive way. The task should have the possibility to save the actual resource state and to close the resource correctly before releasing it. A quasi-preemptive approach was invented by Baunach and is called *dynamic hinting*. In dynamic hinting, a hint is sent to the task that causes a priority inversion. The hint includes the priority of the requested task. With this information, the task decides if to release the resources earlier or not.
To get a hint, it is necessary to find all resources that causes a priority inversion. This critical resources $\text{crit}(t) \subset R$ of a task t are defined in Equation (4.1).

$$\text{crit}(t) := \{R_t | \exists r \in R, w(r) > P_t\} \tag{4.1}$$

The set includes resources that are assigned to task t that has a lower base priority than the tasks that are requesting a resource. Therefore, the hint is sent to task t according to the following condition:

$$\text{hint}(t) := \begin{cases} w(r) & \text{if } r \in \text{crit}(t) \\ \emptyset & \text{otherwise} \end{cases} \tag{4.2}$$

The hint includes the information about the requester's priority to decide if collaborating or not. Figure 4.2 shows an example of the dynamic hinting on a multi-core environment. Suppose that three tasks are running on three different cores. Task L is the first one that requests the resource and therefore gets the resource. At time 1 task M requests the same resource, but the allocation is refused. Because of the priority inversion, task L receives a hint and has its priority boosted. The hint is handled, but the task decides to ignore it. Hence, its priority remains boosted. Later, the task H on the core c_2 requests the same resource. One more time, task L receives a hint, but this time it decides to collaborate. It executes the tail for the component

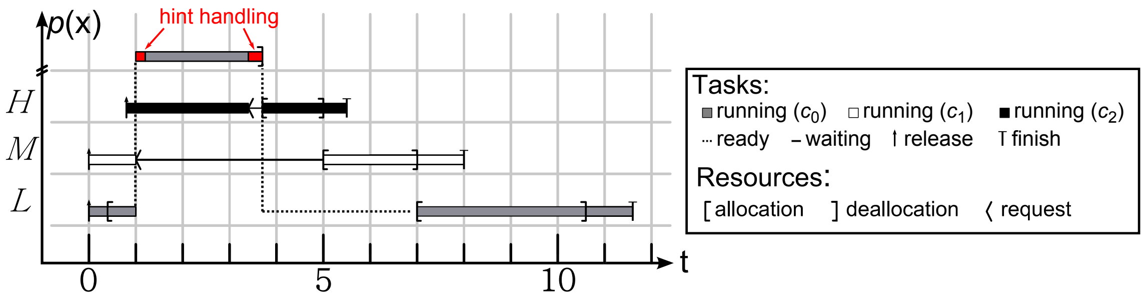


Figure 4.2. – Hinted Multiprocessor Priority Ceiling Protocol example.

and releases the resource. The task with the highest priority allocates the resource and resumes its work. The priority of task L is reduced back to its base priority, because the resource is not assigned to it anymore. At time 5 the resource is released by task H and assigned to task M considering that it has a higher priority than task L. As soon as task L has the possibility to resume, it reallocates the resource and proceeds with its execution.

4.3. Resource Manager

Only a few resource protocols give explanations on how the protocol should be implemented. There are many solutions and they usually depend on the underlying architecture. I used an architecture with a non-shared memory. Hence, the communication between cores is only possible over global semaphores that are offered by the resource manager. In this implementation, the resource management is divided into two layers as shown in Figure 4.3. The upper layer,

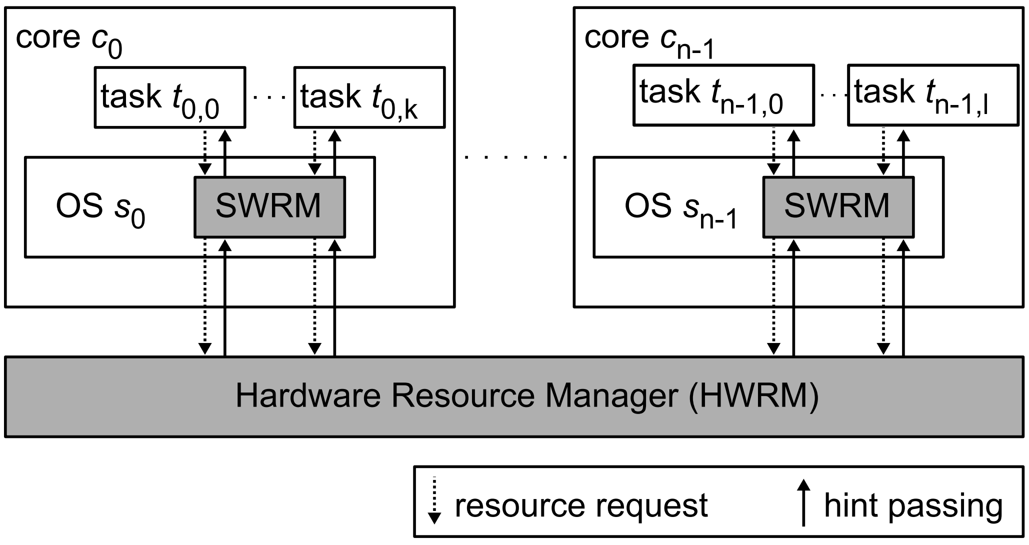


Figure 4.3. – The hierarchy of the implemented Hinted Multiprocessor Priority Ceiling Protocol (HMPCP).

the Software Resource Manager (SWRM), is implemented into the operating system and the lower level, the HWRM, into the hardware. The HWRM is used as a normal MCU-component, like a timer. Making the HWRM as dynamically as possible is only achievable if the HWRM has no knowledge about tasks. The number of tasks on a core, with a few exceptions, is never constant. This means, that the HWRM would have to use a dynamic structure for handling all tasks, or to reserve so many places as the operating system could manage. This approach would use to much space on the hardware. Thus, the HWRM has no knowledge about tasks. To reduce programming errors, the access to the HWRM is always done from the local SWRM on every core implemented into the kernel. The hint from the HWRM is received by the SWRM and then is forwarded to the associated task.

4.4. FPGA Implementation Details

MCUs often do not have a protection for memory and peripherals, because no MMU or MPU is implemented. This leads to the disadvantage that an incorrect software could bring the system into a failure state or a peripheral is accessed without an assigned semaphore. For this reason, the programmer has to pay more attention to avoid software errors. The used architecture MSP430 does not implement memory or peripherals protection. Now, the HWRM adds a protection mechanism for the shared peripherals that are protected by the semaphores. This can be reached by using a crossbar switch. Figure 4.4 shows the general structure of the HWRM. The HWRM has, for each core, an own peripheral bus and additionally an own register set. This allows every core to use the register concurrently. Moreover, the bus is not shared between cores and the resource manager. The advantage is that every core could add non-shared peripherals which are accessible only from its peripheral bus. Another advantage is that for the usage of the peripheral bus no bus synchronization mechanism is needed. The resource manager manages all core operations concurrently and assigns the resource to the core, according to the implemented resource management protocol. If a resource is assigned to a core, the appropriated resource is connected via crossbar to the core. The crossbar switch protects against the connection of more than one core to a peripheral, therefore a bus access conflict between cores can never occur. The resource manager could have more resources ($m := \#R$) than peripherals ($k :=$ number of peripherals), since some resources are not related to a peripheral, but used for task synchronization over cores.
The resource management protocol is implemented into the HWRM. To manage the resources, the resource manager implements an array Q_r for every resource with the length equal to the number of cores, as shown in Figure 4.5. Every core has an exact position in the array it array is internally handled as a priority queue. The array item is the base priority P_t of the task t that will receive the resource. Section 4.5 explains which task should be added to the waiting array, because this array handling is managed by the operating system $s_i \in S$. The core with

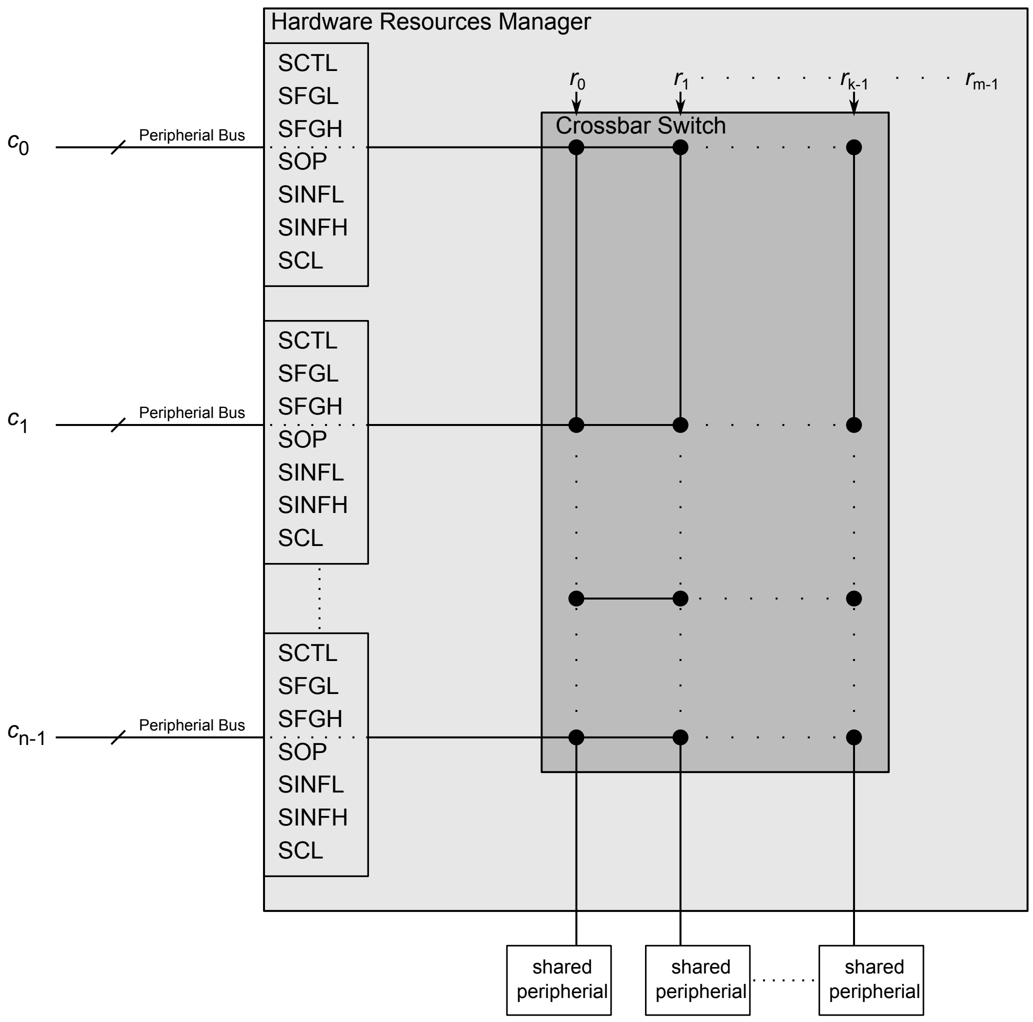


Figure 4.4. – Structure of the Hardware Resource Manager (HWRM).

the highest item, consequently the highest prioritized task that waits for the resource, is going to be handled next for this resource. If the HMPCP assigns the resource to a core, the core is

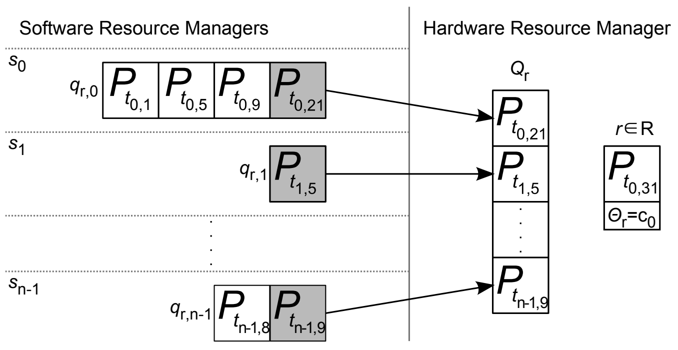


Figure 4.5. – Hardware array and software queue for resource management.

the core-resource owner Θ_r of the resource r. Moreover, the task's priority P_{σ_r} of the assigned task is used for the protocol management.

Registers

Every core that uses the HWRM has its own register set, as shown in Appendix A.
A register that can be found at almost every peripheral is a configuration register. In this component, it is called Semaphore Control Register (SCTL). A flag (IFG) shows if an interrupt is pending. The interrupt flag could be polled by software or an ISR is called. The ISR is only called if this is enabled by the IE flag. Many peripherals on the MSP430 clear the interrupt pending flag automatically before the ISR is called. In my resource manager, this must be done manually by setting the EOI flag. This flag is necessary to let the resource manager know when the interrupt was finally handled. Moreover, there is a flag for synchronization of the whole system at startup. The tasks have to announce their base priority for every used resource by using the Semaphore Ceiling Register (SCL). This is necessary for the resource management protocol. After all tasks announced their priority, the operating systems notifies this to the HWRM by setting the CF flag in the SCTL register. Afterwards, it waits for the flag until it is read as set. The flag is going to be read out as a 1 if all cores have written a 1 into it. This synchronization mechanism guarantees that all ceiling priorities are set correctly, consequently the resource management protocol works correctly with all known priorities.
The registers Semaphore Flag Low Register (SFGL) and Semaphore Flag High Register (SFGH) are used to show if the resource r is assigned to the core c, $(c \in \Theta_r)$.
Semaphore Operation Register (SOP) is the register for handling the resource requests and releases. Additionally, this register is used to change the actual priority in the array Q_r or to remove the item from the array. To perform one of the four operations (request, release, overwrite, timeout) with the SOP register, it is necessary to write on the register the operation's code, and then read it, in order to execute the operation. Listing 4.1 makes a resource allocation request. Firstly, the resource number, the task priority and the selected operation are written into the register. Secondly, the register read takes the written information and executes the operation. In case of an allocation operation, the returned value shows if the request was successful or not. This write and read operations must be done atomically. This means, no

```
SOP = ALOC | num << 8 | priority; //Set the register items
if(SOP == 0)                      //Read operation executes the request
  // allocation refused
else
  // allocation successful
```

Listing 4.1 – Semaphore Operation Register (SOP) usage example.

context switches or interrupts may occur. Mostly, resource requests are done in the kernel of an operating system and this is in most cases not interruptible. If the resource allocation is refused, the request is inserted into the array Q_r on the HWRM. The SOP register also allows to overwrite the array item of its core. The overwrite command could evoke a problem. If the item in the array is removed by the HWRM, because the resource is already assigned to the core, the overwrite operation adds a new item into its position in the array. Theoretically, the assigned item has to be overwritten. To eliminate this race condition the resource manager detects an assignment problem and recovers it by removing the assignment and adding the new item to the array. If no assignment problem occurs, the priority of the assignment is updated. The timeout operation is very similar to the overwrite operation; a race condition could also occur. The timeout operation should remove the item in the array, but if the resource is already assigned to the core, the resource manager has to recover this. It releases the assigned resource and assigns the next allocation according to the management protocol. A further operation that the HWRM offers is the deallocation operation. It is used to release the assigned resource. For the interrupt handling two registers are used: Semaphore Information Low Register (SINFL) and Semaphore Information High Register (SINFH). This registers contain information about the type of interrupt and the number of the resource for which the interrupt is thrown. Moreover, they contain information of the maximum ceiling priority H and the ceiling priority $c(r)$ of the resource r for calculating the boost priority. A requester-priority field is also included in the register, which is needed for hints.

Resource request

The HWRM has to manage any resource request, including concurrent requests from different cores. Listing 4.2 is the Verilog implementation to handle resource requests. The `generate` instruction indicates the compiler to generate this piece of code for every resource, consequently this code is calculated simultaneously for every resource. The iteration marks all cores which make a request for the resource in the `index_request` array. Moreover, it searches the core with the highest requester-priority. This is important, because only the highest priority requester gets the resource, according to the resource management protocol. Lines 14 to 16 define the condition which the request must fulfill to get the resource immediately. As part of the condition, the priority has to be greater than the `ceiling_core` priority. It represents the ceiling priority of all resources that are assigned to cores excluding the core itself. The priority verification for the core itself, is already done on the software level. If all cores request the same resource at the same time, the highest core number has the highest priority. Finally, the core that has the highest request priority and a successful condition, gets the resource immediately. All other requests are added to the waiting array. They are going to be handled by the resource manager later on.

```
generate
for(per_gen=0; per_gen < `R_NR; per_gen=per_gen+1) begin
  always @ * begin : freealloc_or_request
    reg [`C_NR-1:0] index_alloc;
    reg [`C_NR-1:0] index_request;
    index_alloc = 'h0;
    index_request = 'h0;

    // Search the core with the highest priority which does a request
    for (core_inst=0; core_inst < `C_NR; core_inst=core_inst+1) begin

        if (request_cmd[core_inst][per_gen]) begin         //allocation command
          index_request = index_request | (1 << core_inst);//save every request
          if (ceiling_core[core_inst] < sop_prty[core_inst]//allocation possible
                & (~|sema[per_gen] | |res_releasing[per_gen])
                & alloc_max[core_inst])
            index_alloc = (1 << core_inst);
        end else if(overwrite_cmd[core_inst][per_gen])     //overwrite command
            index_request = index_request | (1 << core_inst);
    end

    // Only one request could allocate the resource at one cycle!
    res_free_alloc[per_gen] = index_alloc;
    res_request[per_gen]    = index_request & ~index_alloc;
  end
```

Listing 4.2 – Finding all resource requests and maximal one resource allocation to a core.

HMPCP resource assignment

If the resource allocation can not be granted immediately, the request, with the requesting task's priority, is inserted into the waiting array. When a resource is released, the next task to be assigned to it must be defined by the resource manager. How to search the next resource, which will be assigned to a core, can be calculated in different ways. A simple way to do this would be to find the highest priority in the waiting array by iterating over all resources and all array items sequentially. This leads to a bad performance, because the complexity is finally $\mathcal{O}(\#R \cdot \#C)$. Therefore, searching for the next resource and core that will be connected together is divided into three simultaneous executing parts. Listing 4.3 searches the highest priority for every resource and for every core that is in the waiting queue and it checks, in case of a request, the requester-priority. The highest priority and the core number are going to be saved for the next step, showed in Listing 4.4. Since only one resource can be assigned to a core per request, according to the HMPCP, the resource with the highest priority is searched. In case of two resources with the same maximum, the resource with the lower number is selected first. The register `res_index` saves the selected resource, which will be assigned to a core next. In the last step, shown in Listing 4.5, every resource has to save the number of the core and the priority that will be connected next with the resource. The core number and priority are saved concurrently for every resource, because the code is in a `generate`

```
generate
for(per_gen=0; per_gen < `R_NR; per_gen=per_gen+1) begin
  always @ (*) begin : search_highest_priority_per_resource
    reg [`C_NR-1:0]  tmp_index;
    reg [6:0]        tmp_max;
    reg [6:0]        req_max;
    tmp_index = 'h0;
    tmp_max   = 'h0;
    req_max   = 'h0;

    for(core_inst=0; core_inst < `C_NR; core_inst=core_inst+1) begin
      if(res_request[per_gen][core_inst]            //inclusive actual requests
           & sop_snum[core_inst][per_gen])begin
        if(tmp_max <= sop_prty[core_inst])begin
          tmp_max   = sop_prty[core_inst];
          tmp_index = 1 << core_inst;
        end
        if(req_max < sop_prty[core_inst])
          req_max = sop_prty[core_inst];
      end else begin
        if(res_queue_valid[per_gen][core_inst]      //queue items
             & (tmp_max <= res_queue[per_gen][core_inst]))begin
          tmp_max   = res_queue[per_gen][core_inst];
          tmp_index = 1 << core_inst;
        end
      end
    end   /* end core_inst */
    // save maximum priority with the number of that core
    max[per_gen] = tmp_max;
    core_index[per_gen] = tmp_index;
    res_request_highest_priority[per_gen] = req_max;
  end
end
endgenerate
```

Listing 4.3 – Find the core number for each resource with the highest priority, which requested the resource and save that into a register with the corresponding priority.

```
always @ (*) begin : next_res_to_handle
  reg [`R_NR-1:0] tmp_index;
  reg [6:0]       tmp_max;
  tmp_index = 'h0;
  tmp_max   = 'h0;

  for(per_inst=0; per_inst < `R_NR; per_inst=per_inst+1) begin
    if(tmp_max < max[per_inst]) begin
      tmp_max   = max[per_inst];
      tmp_index = 1 << per_inst;
    end
  end /* end per_inst */
  // resource number with the highest priority
  res_index = tmp_index;
end  /* next_res_to_handle */
```

Listing 4.4 – Find the resource number with the highest priority in all waiting queues.

```
generate
for(per_gen=0; per_gen < `R_NR; per_gen=per_gen+1) begin
  always @ (*) begin
    res_next_core[per_gen]     = (res_index[per_gen])? core_index[per_gen]: 'h0;
    res_next_priority[per_gen]= (res_index[per_gen])? max[per_gen]       :6'h0;
  end
end
endgenerate
```

Listing 4.5 – Set for every resource the core number which core will get the resource as next.

block. Since, only one resource can be assigned to a core (see Listing 4.4), only one resource (`res_next_core[per_gen]`) gets the identifier of that core. With the separation into three parallel computing parts the complexity is reduced to $\mathcal{O}(\#R + \#C)$, but with a higher area usage in the hardware.

Rise, Fall and Allocation signal

The signals that notify a task of a new allocated resource or of changing its priority according to HMPCP are described in this section. Listing 4.6 is the Verilog implementation for this operation. The HWRM detects a priority inversion and signals the associated core with a rise priority message, which is also a hint message. A rise signal is thrown if some core executes a resource request, which has a higher priority than the actual priority of the assigned task. The higher requester-priority is additionally saved for the hint. After a priority inversion is eliminated due to a timeout, the priority of a task has to be set back to its base priority. The priority can be set back if and only if all priorities in all waiting arrays are lower than the actual assigned resource, thus no priority inversion occurs. The fall signal is only forwarded if the priority was already risen.
After a refused resource request, the request is going to be inserted into the waiting array. If the condition in lines 32 to 38 is true, the resource is assigned to the waiting core in the next cycle. This can happen only if the core contains the highest priority in the waiting array for this resource, see line 33. Further, the current system ceiling priority is lower than the priority of the waiting one and a core releases its resource.
All this three signals are used to synchronize the internal status and they are forwarded to the interrupt section of the HWRM.

```
for(core_gen=0; core_gen < `CORE_NR; core_gen = core_gen + 1) begin
  for (per_gen=0; per_gen < `RM_SEM_SIZE; per_gen=per_gen + 1) begin
    always @ (*) begin : rise_fall_alloc_signal
      reg tmp_rise, tmp_fall;
      reg [6:0] max_priority;
      reg tmp;
      reg core_releasing;

      tmp_rise = 1'b0;
      tmp_fall = 1'b1;
      max_priority = (rise_signal[core_gen][per_gen]) ? rise_priority_dly  : 'h0;
      core_releasing = 1'b0;

      for (per_inst=0; per_inst < `RM_SEM_SIZE; per_inst=per_inst+1) begin

        tmp = |res_request[per_inst]
              & res_priority[per_gen] < res_request_highest_priority[per_inst];
        tmp_rise = tmp_rise | tmp;

        if(tmp & max_priority < res_request_highest_priority[per_inst])
          max_priority = res_request_highest_priority[per_inst];

        tmp_fall = tmp_fall
                   & (res_next_priority[per_inst] < res_priority[per_gen]);
        core_releasing = core_releasing | |res_releasing[per_inst];
      end

      rise_priority[core_gen][per_gen] = max_priority;
      rise_one_shot[core_gen][per_gen] = tmp_rise & sema[per_gen][core_gen];
      fall_one_shot[core_gen][per_gen] = tmp_fall & sema[per_gen][core_gen]
                                         & priority_risen[core_gen][per_gen];
      alloc_one_shot[core_gen][per_gen] =
         res_next_core[per_gen][core_gen] // core,res next one to assign?
         & current_ceiling < res_next_priority[per_gen] // MPCP condition
         & core_releasing            // another core releases the resource
         & ~|res_free_alloc[per_gen] // prevent alloc. while another free_alloc
         & ~res_remove_queue[per_gen][core_gen]// prevent handover at concurrent
         & ~res_overwrite_queue;               // ... timeout or overwrite
    end
  end  /* end per_gen */
end /* end generate cpu */
```

Listing 4.6 – Calculation for the rise, fall and allocation signals.

Interrupt

Interrupts are used by the HWRM to inform the core about a new important message that must be handled by the operating system. The Semaphore Information Low Register (SINFL) and Semaphore Information High Register (SINFH) give the operating system the information from the resource manager. There are three types of interrupts: allocation, rise and fall. The operating system distinguishes between them through the SINFL register. The allocation interrupt is responsible for notifying the core that it receives a resource after waiting for it. The rise interrupt notifies the core that the priority of one of its tasks has been boosted. Additionally, it also notifies the core if a new hint is available for it. The fall interrupt notifies the core when a previously boosted task priority is set back to the base priority. This interrupt can only be thrown if a prior rise interrupt was handled. Furthermore, the HWRM eliminates potential race conditions, e.g. the fall interrupt is pending and the kernel operation releases the resource. After exiting the kernel mode, the interrupt would be executed. Hence, the resource manager detects an already released resource and cancels the fall interrupt.
Listing 4.7 shows the code that checks for the next interrupt. This code is generated for every core. Therefore, a simultaneous interrupt handling of every core is possible. Every core iterates over every peripheral, from line 20 to line 37, to search for the next interrupt. In each iteration the allocation, rise and fall signals are checked. The `alloc_signal`, `rise_signal` and `fall_signal` which are checked in the conditions are signals inhered from the `alloc_one_shot`, `rise_one_shot` and `fall_one_shot` signals, see Section 4.4.
For every kind of interrupt, the highest prioritized peripheral number that has to throw an interrupt is saved. The core can handle only one interrupt at a time, therefore all interrupts must be handled one after another. The most important interrupt must be thrown firstly and this is handled from line 40 to line 54. To throw the interrupts in a useful order, the allocation interrupt has the highest priority followed by the rise and the fall interrupt. The selected interrupt is only thrown to the core if an older interrupt handling was successful. Therefore, if the interrupt was handled successfully by the software, the code has to notify this by setting the EOI flag in the SCTL register. Thus, the HWRM can throw the next queued interrupt. Additionally, a register (`irq_next_num`) is used to save the peripheral number that throws the interrupt to the core. This register is used to set the content in the SINFL and SINFH registers.

```
generate
for(core_gen=0; core_gen < `C_NR; core_gen = core_gen + 1) begin

  always @ (*) begin : next_irq
    reg [7:0] index_rise, index_fall, index_alloc;
    reg [6:0] max_rise, max_fall, max_alloc;

    index_rise = 'hff;
    index_fall = 'hff;
    index_alloc = 'hff;
    max_rise = 'h0;
    max_fall = 'h0;
    max_alloc = 'h0;

    irq_rise[core_gen] = 'h0;
    irq_fall[core_gen] = 'h0;
    irq_alloc[core_gen] = 'h0;
    irq_new[core_gen] = 'h0;

    for (per_inst=0; per_inst < `R_NR; per_inst=per_inst+1) begin
      // search for every event the highest priority
      if (alloc_signal[core_gen][per_inst]
            & (max_alloc < res_next_priority[per_inst])) begin
        index_alloc = per_inst;
        max_alloc = res_next_priority[per_inst];
      end
      if(rise_signal[core_gen][per_inst]
            & (max_rise < rise_priority[core_gen][per_inst]) ) begin
        index_rise = per_inst;
        max_rise = rise_priority[core_gen][per_inst];
      end
      if (fall_signal[core_gen][per_inst]
            & (max_fall < res_priority[per_inst])) begin
        index_fall = per_inst;
        max_fall = res_priority[per_inst];
      end
    end

    // set the interrupt signal according the interrupt priority
    if(index_alloc != 'hff) begin              /* Highest Priority */
      irq_alloc[core_gen][index_alloc] = 1'b1;
      irq_next_num[core_gen] = index_alloc;
      irq_new[core_gen][index_alloc] = eoi_acc[core_gen];
    end else if (index_rise != 'hff) begin
      irq_rise[core_gen][index_rise] = 1'b1;
      irq_next_num[core_gen] = index_rise;
      irq_new[core_gen][index_rise] = eoi_acc[core_gen];
    end else if (index_fall != 'hff) begin    /* Lowest Priority */
      irq_fall[core_gen][index_fall] = 1'b1;
      irq_next_num[core_gen] = index_fall;
      irq_new[core_gen][index_fall] = eoi_acc[core_gen];
    end  else begin
      irq_next_num[core_gen] = 'h0;
    end
  end
end
endgenerate
```

Listing 4.7 – Search the next interrupt to throw independently for each core.

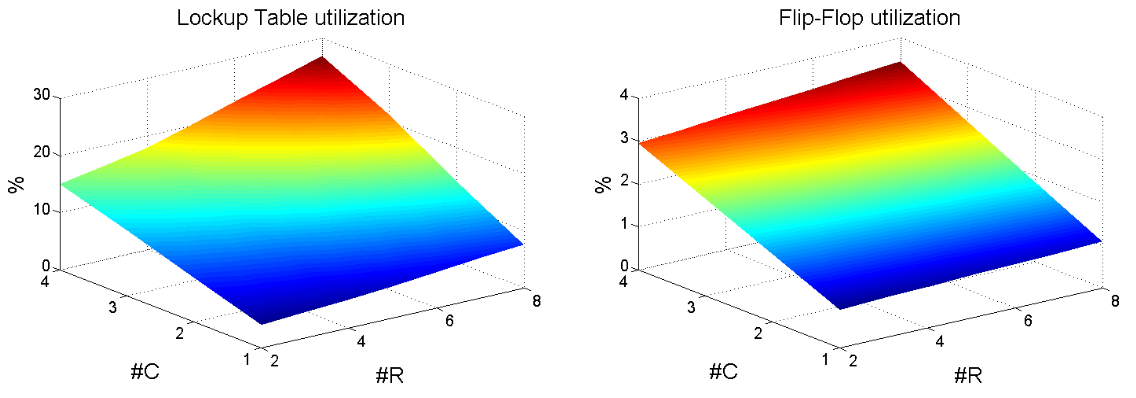
Resource Utilization

I evaluated the number of LookUp Tables (LUTs), Flip-Flops (FFs), Block Random Access Memories (BRAM) and Digital Signal Processors (DSPs) of the whole system with different numbers of instantiated cores and resources. All the measurements are taken from the implementation report in the Vivado 2014.4 for the Xilinx Artix-7 XC7A100T (Speed Grade-1) FPGA. Table 4.1 summarizes the space utilization for each part of the FPGA for different configurations. BRAM and DSP are not needed by the HWRM, but specific to each core. Therefore, their space usage remains constant if the number of cores is constant.

Configuration	LUT	FF	BRAM	DSP
$\#C$=1, $\#R$=2	2612 (4.12 %)	1141 (0.90 %)	15 (10.74 %)	1 (0.42 %)
$\#C$=1, $\#R$=4	3113 (4.91 %)	1217 (0.96 %)	15 (10.74 %)	1 (0.42 %)
$\#C$=1, $\#R$=8	4844 (7.64 %)	1369 (1.08 %)	15 (10.74 %)	1 (0.42 %)
$\#C$=2, $\#R$=2	4901 (7.73 %)	2003 (1.58 %)	29 (21.48 %)	1 (0.83 %)
$\#C$=2, $\#R$=4	5915 (9.33 %)	2130 (1.68 %)	29 (21.48 %)	2 (0.83 %)
$\#C$=2, $\#R$=8	8939 (14.10 %)	2371 (1.87 %)	29 (21.48 %)	2 (0.83 %)
$\#C$=3, $\#R$=2	7227 (11.40 %)	2866 (2.26 %)	44 (32.22 %)	3 (1.25 %)
$\#C$=3, $\#R$=4	8781 (13.85 %)	3043 (2.40 %)	44 (32.22 %)	3 (1.25 %)
$\#C$=3, $\#R$=8	13 365 (21.08 %)	3373 (2.66 %)	44 (32.22 %)	3 (1.25 %)
$\#C$=4, $\#R$=2	9478 (14.95 %)	3740 (2.95 %)	58 (42.96 %)	4 (1.67 %)
$\#C$=4, $\#R$=4	11 292 (17.81 %)	3956 (3.12 %)	58 (42.96 %)	4 (1.67 %)
$\#C$=4, $\#R$=8	17 061 (26.91 %)	4387 (3.46 %)	58 (42.96 %)	4 (1.67 %)
Xilinx XC7A100T	63 400	126 800	135	240

Table 4.1. – Resource utilization with different configurations.

The FPGA resources needed by LUTs and FFs, on the contrary, depend on both number of cores and of resources, as shown in Figure 4.6. Since the HWRM needs more registers if more



(a) LookUp Table (LUT) utlization. **(b)** Flip-Flop (FF) utlization.

Figure 4.6. – Diagrams of the FPGA resource utilization with different configurations.

resources are instantiated and for every core, a new register set must be created. Therefore, the space needed will increase for every added core or resource.

Performance

The performance measurements are done for the same FPGA and with the same development environment as for the Resource Utilization. The development environment gives the information about the Worst Negative Slack (WNS) into a report file. The WNS demonstrates the remaining time between the longest path time on the system and the constrained clock time. The contained clock in my environment is the MCLK. It is an 8 MHz clock, which corresponds to a period of 125 ns. Table 4.2 shows the WNS with different environment configurations and the resulting maximal achievable frequency for the MCLK. Moreover, it shows that for some configurations, with a negative WNS, the timing constraints cannot be fulfilled with the targeted 8 MHz frequency.

Configuration	WNS	max freq.
#C=1, #R=2	81.57 ns	23.03 MHz
#C=1, #R=4	66.94 ns	17.22 MHz
#C=1, #R=8	23.86 ns	9.89 MHz
#C=2, #R=2	69.74 ns	18.10 MHz
#C=2, #R=4	41.42 ns	11.96 MHz
#C=2, #R=8	0.26 ns	8.02 MHz
#C=3, #R=2	59.58 ns	15.29 MHz
#C=3, #R=4	39.19 ns	11.65 MHz
#C=3, #R=8	−5.92 ns	7.64 MHz
#C=4, #R=2	49.97 ns	13.33 MHz
#C=4, #R=4	24.89 ns	9.99 MHz
#C=4, #R=8	−17.01 ns	7.04 MHz

Table 4.2. – Performance results with different configurations ($f_{MCLK} = 8$ MHz).

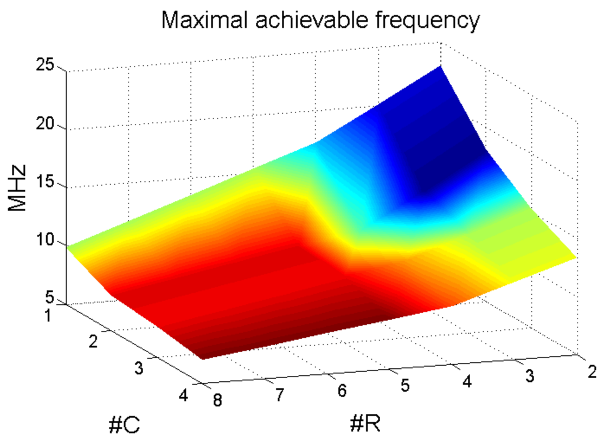


Figure 4.7. – Maximal achievable frequency with different configurations.

Figure 4.7 shows the results in a graphical representation. It shows that the maximal achievable frequency decreases rapidly with only a few cores and/or resources more. The cause of this behavior is that the implementation is totally optimized for speed, because all its work is done in only one cycle. Therefore, it is necessary to iterate over all cores and resources in only one cycle and this needs the use of combinational logic that slows down the maximum achievable frequency. Furthermore, the HWRM is a centralized module. With many instantiated cores and peripherals that are all connected to this single point, the paths increase and therefore the WNS is reduced.

4.5. SmartOS Implementation Details

The Software Resource Manager (SWRM) is implemented in the operating system $s_i \in S$ and extends local resource management with global resources. It works as a middleware, because the local resources are going to be handled locally and in the case of global resources, the commands are forwarded to the HWRM. The HWRM has for every resource and for every core only one room in the array for all tasks. Therefore, the resource manager on the software side must insert the correct task into the HWRM array, as shown in the example of Figure 4.5. To manage this, for every resource $r \in R$ a local priority queue q_r is needed, which is ordered by task priority. The head of the queue q_r is inserted into the waiting array Q_r on the HWRM. At startup the HWRM and the internal structures have to be initialized as in Listing 4.8. First, every task t has to announce its base priority P_t to the HWRM for every resource it might use. Afterwards, for every global resource, the locally used structure is going to be initialized. The counter n_r, which counts the number of reentrant resource allocations from the same task, is set to zero. The owner of each resource is set to "no task" and the locally waiting queue is cleared. Finally, the interrupts have to be enabled for receiving the interrupts from the HWRM. The SmartOS Resource Control Block (RCB) is extended with a field to indicate the global resource number. If the field is set to 0xFFFF, the resource is configured as a local resource and is therefore handled with the local resource manager algorithms. In the case of a global resource, the syscalls (see Appendix C) are handled by the global resource manager algorithms, which are shown in the next sections.

```
HW_RM_INIT(){
  /* all tasks set their ceiling priorities */
  while($r \in R$){
    $n_r := 0$ // reentrant allocation counter
    $\sigma_r := \emptyset$ // owner
    $q_r := \emptyset$ // priority waiting queue
  }
  /* enable hardware resource manager interrupts */
}
```

Listing 4.8 – Initialization of the Hardware Resource Manager (HWRM) and the initialization of the data structures in the Software Resource Manager (SWRM).

Resource allocation

Before a task can use a resource, it must allocate the resource. Listing 4.9 shows the pseudo code for the global resource allocation, executed in kernel mode. Therefore, the code cannot be interrupted. There are three different possible paths. In the first case, the resource r is just assigned to task t, because it is the task-resource owner σ_r of resource r therefore, nothing has to be done in the HWRM. The counter n_r only needs to be incremented to handle the releasing operation correctly.
In the second case, the software resource allocation queue is empty, hence no other task is waiting for the global resource r in the current core. Before forwarding the allocation command to the hardware, the hardware has to check if, according to MPCP, the allocation command is permitted or not. In the lines 9 to 11, the highest base priority of all tasks, without itself, in the core which are assigned to some resources are searched. In the case that the base priority of the requester is higher than the calculated maximum, the allocation command is sent from the operating system to the HWRM, see line 14. Otherwise, an overwrite command is sent to the resource manager, see line 16. The overwrite command only inserts the task into the resource manager without allocating it immediately. This process is necessary, because the HWRM checks only the ceiling priorities without the requester core to allow nested locks.

```
 1 allocation ($t$, $r$) {
 2   ret := 0
 3
 4   if($\sigma_r == t$){ //r is just assigned to t
 5     $n_r := n_r$ + 1;
 6     return
 7   } else if($q_r == \emptyset$){
 8     max := 0
 9     while($s \in R$)
10       if(max < $P_{\sigma_s} \wedge \sigma_s \neq t \wedge \sigma_s \neq \emptyset$)
11         max = $P_{\sigma_s}$
12
13     if(max < $P_t$)
14       ret := ALLOCATION($P_t$, $r$);
15     else
16       ret := OVERWRITE($P_t$, $r$);   // insert into the hardware queue
17   } else {
18     if(P_{head(q_r)} < $P_t$)
19       ret := OVERWRITE($P_t$, $r$);   // overwrite the actual queue item
20   }
21
22   if(ret $\neq 0$)
23     $\sigma_r := t$;
24   else {
25     $q_r = q_r \cup \{t\}$;
26     supsend($t$);
27   }
28 }
```

Listing 4.9 – Resource request handling for a global resource.

In the last case, the waiting queue is not empty. In the case that the requester's priority P_t is higher than the head of the local queue q_r, an update is needed in the HWRM, with an overwrite command. In the lines 22 to 27, the return values from the allocation and overwrite commands, which come from the HWRM, are checked. If the commands return a zero, the resource is not assigned to the task and therefore, the task is inserted into the local queue and is moved to suspended state. It will stay in this state until the resource is assigned to the task. Otherwise, task-resource owner σ_r is set to t and the task t remains executing. Not only the allocation operation could return a positive resource allocation flag, but also the overwrite operation. This is needed, because other cores on the system remain operative. It could happen that the waiting resource request is being assigned to the requesting core, while the core is running in the kernel mode, where interrupts are disabled. The assignment interrupt is not necessary anymore, because the return value of the overwrite command just shows the resource allocation.

Resource deallocation

After a successful assignment of a resource the task has the possibility to release the resource. Listing 4.10 demonstrates the pseudo code for the release of resource r. As in the allocation, the release also runs in kernel mode. Before releasing the resource, the function checks how often the resource was allocated by the same task. The resource is deallocated if, and only if, the number of allocations was equal to the number of releasing, for the same resource and for the same task. If this condition is true, the release command is sent to the HWRM and the owner of the resource is set to empty. Then, the rise bit in the bit-field r_t is cleared, that shows an early received rise signal from the resource manager for the resource r. If no flag in the bit-field r_t is set anymore, the active priority $p(t)$ of task t is set to its base priority P_t, because it could happen that the priority is boosted because of a caused priority inversion.

```
deallocation($t$,$r$){
  $n_r := n_r$ - 1;
  ASSERT($n_r \geq 0$);
  if($n_r == 0$){
    DEALLOCATE($r$);
    $\sigma_r := \emptyset$

    $r_t$ &= ~(1 << id($r$));
    if($r_t == 0$)
      $p(t) := P_t$;
  }
}
```

Listing 4.10 – Resource releasing handling of a global resource.

Resource timeout

The operating system offers the possibility to limit the maximum waiting time of a task for a resource request through a timeout or a deadline. In the meantime, the task remains in suspended state and cannot execute other code. If the maximum time is elapsed, the pseudo code in Listing 4.11 is executed from the operating system, hence in kernel mode. First of all, the task t is removed from the local waiting queue q_r and the task is set to ready state. Afterwards the HWRM is notified about this new event. Figure 4.8 demonstrates the used operation, depending on the state of the local queue q_r and the position of the the task in it. If the timeout request is not the head of the local waiting queue, nothing needs to be communicated to the HWRM. However, if it is the head and there is no request that could replace the timed out request, the operating system uses the timeout command to remove it from the HWRM's waiting array Q_r. If other tasks also wait in the local queue q_r, the overwrite command is used in order to replace the request in the global array Q_r. The overwrite command returns a value to show if the resource is already assigned to the core. If the resource has just been assigned to the core, the head task in the queue is switched to ready state, because it is

```
timeout($t$,$r$){
  old_head := head($q_r$);
  $q_r = q_r \setminus \{t\}$;
  resume($t$)

  if($q_r == \emptyset$)                // only task t was in the queue
    TIMEOUT($r$);
  else if (old_head == $t$){            // task t was the head of the queue
    s := head($q_r$);
    ret := OVERWRITE($P_s$, $r$);       // notify the HWRM about the new head
    if(ret $\neq$ 0){                   // resource is just assigned
      $\sigma_r := s$;
      $q_r = q_r \setminus \{s\}$;
      resume($s$)
    }
  }
}
```

Listing 4.11 – Timeout handling for a timeouted global resource request.

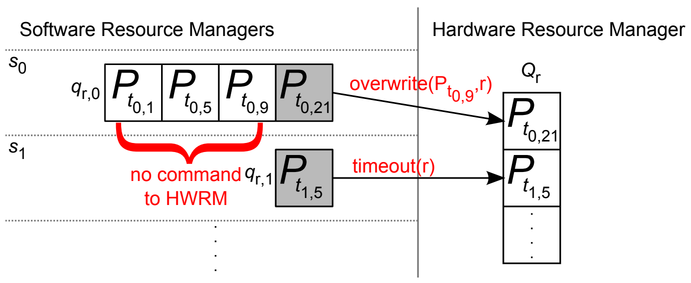


Figure 4.8. – Used operations of timed out tasks.

the new owner of the resource and it is removed from the waiting queue.

Interrupt handling

An interrupt is a useful concept to interrupt the execution flow for an important message from peripherals. The HWRM has one interrupt that is used for three kinds of messages. The highest prioritized messages notify the assignment of a resource to a core. If a resource request for a resource r was refused, the task's priority is inserted into the HWRM array Q_r. Afterwards, when the resource is assigned to the requested core, according to HMPCP, the operating system will be notified with the allocation interrupt and handle it as in Listing 4.12. The hardware

```
ISR_ALLOC(){
  $r := r_{ISR}$;    // read from register
  $t :=$ head($q_r$)

  $\sigma_r := t$;
  $q_r := q_r \setminus \{t\}$;
  resume($t$);
}
```

Listing 4.12 – Allocation interrupt handling.

provides in a register which resource is currently assigned to the core. The head task of the local waiting queue q_r is going to be resumed for its execution. Moreover, it is removed from the local waiting queue and the resource owner is set to the new task. The second message that the HWRM sends to the operating system is that a priority inversion occurs with the associated hint information. This interrupt is handled as shown in Listing 4.13. The resource number is read out from the SINFL register. The owner task σ_r of the resource r is going to be boosted according to the HMPCP, but only for the first rise interrupt. Furthermore, it uses a bit-flag r_t for every task t that shows for which resource the task is risen. This is important to handle, because the HWRM has no knowledge about tasks. Therefore, the rise and fall interrupts are also sent more than once to a task, for every assigned resource an interrupt. At the end of this interrupt handling, the hint is sent to the task that causes the priority inversion. The hint

```
ISR_RISE_HINT(){
  $r := r_{ISR}$;    //read from register
  $t := \sigma_r$
  if($r_t == 0$)
    $p(t) := H + 1 + c(r)$ //boosted priority
  $r_t$ |= 1 << id($r$);
  send_hint(t, hint_infos); //hint_infos read from registers
}
```

Listing 4.13 – Rise and hint interrupt handling.

```
ISR_FALL(){
  $r := r_{ISR}$;    //read from register
  $t := \sigma_r$

  $r_t$ &= ~(1 << id($r$));
  if($r_t == 0$)
    $p(t) := P_t$   //set to base priority
}
```

Listing 4.14 – Fall interrupt handling.

includes information such as the priority of the resource requester.
The timeout operation could have the side effect that a priority inversion is eliminated. This leads to the reduction of the task's boosted priority to its base priority as shown in Listing 4.14. The resource is going to be removed from the rise indication bit-flag r_t. If no rise flag is set anymore, the priority of the task is set back to its base priority. It means that the task does not cause a priority inversion anymore with its assigned resources.

5. Evaluation

This chapter demonstrates the evaluation of the implemented Hinted Multiprocessor Priority Ceiling Protocol. The first section introduces the test-bed used for the implementation and for the time and functional measurements. Section 5.2 starts with some time behavior measurements for the implemented HWRM. In addition, it shows some test benches of the hardware implementation. These test benches test the functional behavior. The chapter concludes with a use case of two shared resources, which simulated a common resource allocation pattern.

5.1. Test-bed

For the test-bed, the development board Nexys^{TM}4 DDR Artix-7 FPGA Board[2] from Digilent is used. On the board an Artix-7 FPGA from Xilinx is mounted. For synthesis, the tool Vivado 2014.4 from Xilinx is used. Xilinx offers a free available version called WebPack that can be downloaded for free from their website[3]. The development board has many input and output pins that are used for time measurements. Furthermore, the FPGA involves a huge number of LUTs, FFs and BRAM. The internal memory is used for program and data memory. Figure 5.1 shows the general structure of the test environment. The processor is a symmetric quad-core, what means that all four cores are identical. The cores are openMSP430 cores, each one using its own data and program memory. Therefore, there is no need for memory partitioning. Hence, there is no need for concurrent bus access. The peripheral bus of each core is connected with the resource manager. There are some peripherals between the cores and the peripheral bus, which are local and accessible only for the bus owner. For every core there is a timerA peripheral that can be used by the operating system. Moreover, each core has its own multiplier and watchdog. The SFRs are special registers in the MSP430 architecture. They include the enable and disable register for global interrupts and NMIs. OpenMSP430 extends this with a register in which the number of cores on the system is stored, as well as the core's own number. On the development board, a 100 MHz oscillator is mounted. This oscillator is divided in a 8 MHz clock, corresponding to a period of 125 ns, and is used for the whole system. The openMSP430 includes two components shared by the whole system: the GPIO and the UART. Both components are protected by the HWRM and are only accessible if

[2] `http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,719,1337&Prod=NEXYS4DDR`
[3] `http://www.xilinx.com/support/download.html`

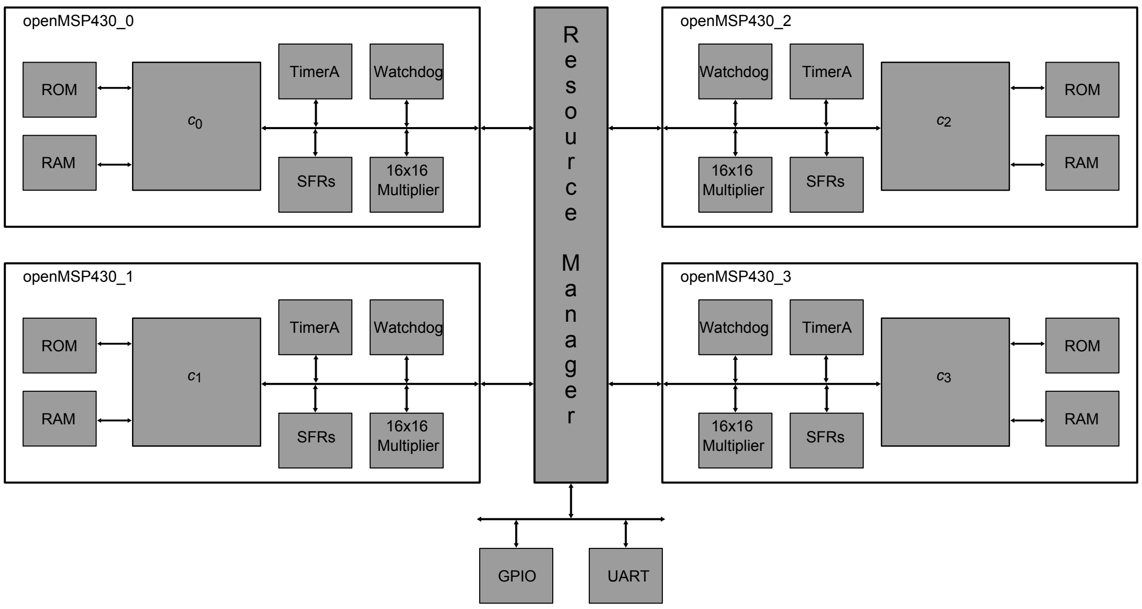


Figure 5.1. – Test enviroment for the evaluations.

the resource is assigned to the core.
The USB-ISS[4] device is used as a debug interface. This interface could be used as a USB to SPI, I^2C, GPIO or UART converter. The openMSP430 debugger could be used with UART and I^2C. In case of UART, for every core an own interface is needed. The I^2C is a bus, which can be used to communicate with all cores with only two lines: SCL and SDA. Every core is assigned with a unique address and this makes it possible for the debugger to communicate with a sole core. Moreover, there is a broadcast address, where all cores receive the commands.
The debugger does not only allow to debug the program on the core, but also to upload the ELF-file into the program memory. Besides, it is also possible to upload the program by using the broadcast address. Thus, all program memories are written at the same time. Certainly, all program memories have the same content, but the core uses the core number from the SFR register to decide which parts in the program should be run by it.
For the measurements, analog measures are done with the *Lecroy WaveAce 102* oscilloscope and digital logic measures with the *PicoScope 2205 MSO* oscilloscope.

5.2. FPGA standalone Test

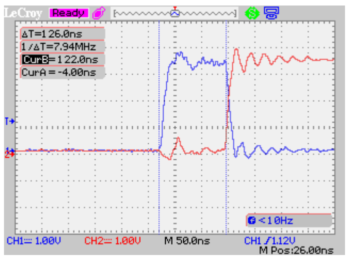
This section shows the test outcomes of the HWRM without an operating system. Firstly, it shows the timings of the allocation and release operation. Afterwards, some test benches for the functional operations are shown. It starts with the resource assignment, like the allocation

[4]`http://www.robot-electronics.co.uk/htm/usb_iss_tech.htm`

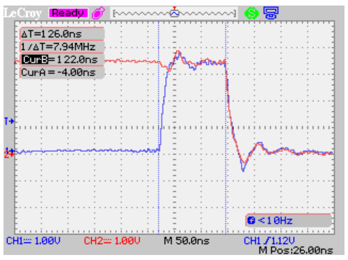
operation or the assignment of a queued resource request by the HWRM. Further, the timeout and overwrite operators are tested. This shows the assignment recovery of the HWRM to prevent race conditions. The section is concluded with tests for the interrupt throwing.

Time Measurements

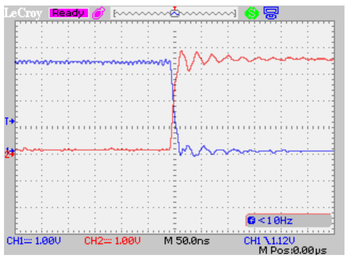
The HWRM has the job to assign and dissociate a resource to and from a core. The core sends the command to the resource manager via a peripheral register. After receiving a request or a release command, the HWRM is going to manage the command immediately. Figure 5.2a shows the resource assignment of a free resource to a requested core. It shows that the resource is assigned to the core after about 126 ns, which is the period of one clock cycle. This means that a free resource is assigned to the core in the next cycle. For the case of a resource release, the resource is deallocated from a core after about 126 ns. Figure 5.2b shows the deallocation of the resource after a release command from the core. Moreover, the resource is deallocated from the core after one clock cycle, as the allocation of a free resource.
If tasks on cores are blocked for execution because of a refused resource allocation, the requester cores are inserted into the array on the HWRM. This could happen if the allocation is not possible, because of the resources manager rules. After releasing an assigned resource, the resource manager manages the next assignment of a queued resource request. Figure 5.2c shows the assignment handover of a released resource to a queued resource request. One cycle after the release command, the resource r_0 is deallocated from the core, as shown in Figure 5.2b. Concurrently, the queued request for resource r_1 is assigned to the requested core.



(a) Free resource allocation (blue = request red = allocation).



(b) Assigned resource deallocation (blue = release red = allocation).



(c) Resource handover (blue = resource r_0 red = resource r_1).

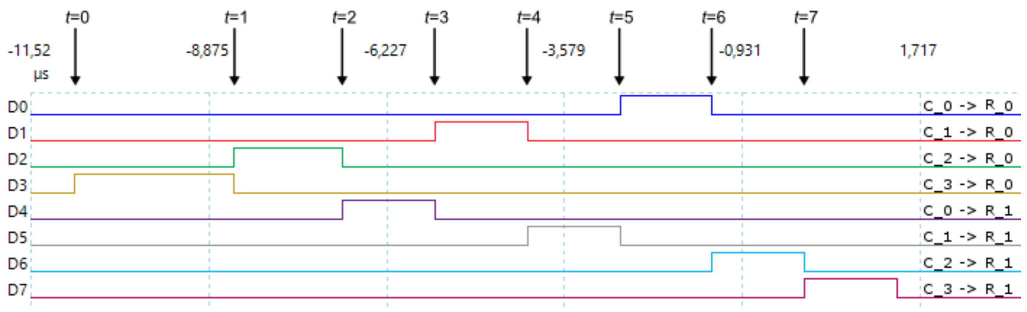
Figure 5.2. – Time measurements of resource allocations/deallocations on the HWRM.

Resource assignment

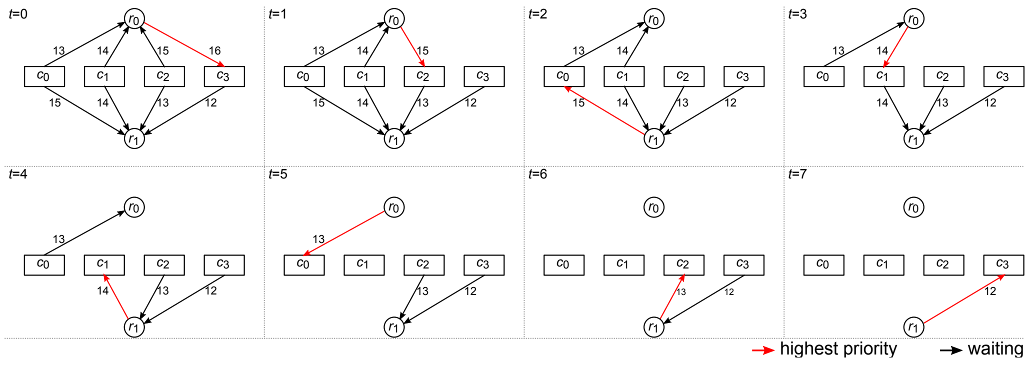
On a multi-core environment, resource requests could be done, in contrary to a single-core environment, from more than one core. Therefore, the resource manager must be able to manage the resource requests concurrently. Listing 5.1 demonstrates a concurrent resource request at lines 6 and 7. The measured assignments of the resource are shown in Figure 5.3a and Figure 5.3b explains the same with a resource allocation graph. Firstly, the highest-priority request gets the resource. After core c_3 releases resource r_0, the HWRM has two options to assign the next resource to a core, because two requests have the same priority. One option is to assign resource r_0 to core c_2 and the other option is to assign resource r_1 to core c_0. In my implementation, the resource r_0 is assigned to core c_2, because of the lower resource number. The lower the resource number, the higher the priority, in the case of equal task priorities which perform a request. After executing this assignment, the second option is going to be used. In the next step, core c_0 releases resource r_1, then the resource r_0 is assigned to core c_1 by the HWRM. Similar as before, the lower resource number is assigned first. Here it is important that both resources r_0 and r_1 are not assigned to the core c_1, although both resource requests priorities are equal. The reason for this is that this access pattern is not possible from a single task, because the task sleeps if the resource request is refused. Therefore, the code simulates the resource request from two different tasks in an operating system. The rest of the code is straightforward.

```
 1 /* initialization */
 2 SET_CEIL(0, 16);
 3 SET_CEIL(1, 15);
 4
 5 /*C_0*/           /*C_1*/           /*C_2*/           /*C_3*/
 6 ALLOCATE(0, 13);  ALLOCATE(0, 14);  ALLOCATE(0, 15);  ALLOCATE(0, 16);
 7 ALLOCATE(1, 15);  ALLOCATE(1, 14);  ALLOCATE(1, 13);  ALLOCATE(1, 12);
 8                                                       DEALLOCATE(0);
 9                                     SCTL |= EOI;
10                                     DEALLOCATE(0);
11 SCTL |= EOI;
12 DEALLOCATE(1);
13                   SCTL |= EOI;
14                   DEALLOCATION(0);
15                   SCTL |= EOI;
16                   DEALLOCATE(1);
17 SCTL |= EOI;
18 DEALLOCATE(0);
19                                     SCTL |= EOI;
20                                     DEALLOCATE(1);
21                                                       SCTL |= EOI;
22                                                       DEALLOCATE(1);
```

Listing 5.1 – Example of a concurrent resource request and the resource assignment depending of the priority.



(a) Resource assignment output.



(b) Resource allocation graph.

Figure 5.3. – Resource assignment over the time of the executed code in Listing 5.1.

Listing 5.2 shows an example of nested locks and the resource assignment of more than one resource at a time. The output of the measures are shown in Figure 5.4. At line 8, the resource r_0 is assigned to the core c_1. In the next lines, the core c_0 requests the resources r_3 and r_2, both are immediately assigned to it. The resource r_3 is assigned to the core because the requested priority is higher than the ceiling priority of resource r_0. Therefore, the priority of the requested resource is higher than the current system ceiling priority M. The allocation of resource r_2 is also granted although the priority is not higher than the current system ceiling priority M. However, the HWRM only checks the ceiling priority of all other cores, except the own core. That means, the ceiling priority excluding the core itself is 1, from the allocated resource r_0 to core c_1. This access pattern is simulating a nested resource request by one task. If the request is done by another task on the same core, this is handled by the local SWRM that prevents a resource request to the HWRM. At line 11 the core c_1 produces one more time a resource request, but now the allocation is refused. The reason for that is that the ceiling priority on all cores excluding itself is higher than the requested priority. Therefore, the task must wait and after a defined timeout, if it still can not allocate the resource, the resource request is removed. Figure 5.4 shows the output of the assigned resources to the cores and that the resource r_1 is never assigned to the requested core.

```
/* initialization */
SET_CEIL(0, 1);
SET_CEIL(1, 2);
SET_CEIL(2, 3);
SET_CEIL(3, 4);

/* C_0 */                        /*C_1*/
                                 ALLOCATE(0,1); /*allocation*/
ALLOCATE(3,3); /*allocation*/
ALLOCATE(2,3); /*allocation*/
                                 ALLOCATE(1,2); /*allocation refused*/
                                 TIME_OUT(1);
DEALLOCATE(3);
DEALLOCATE(2);
                                 DEALLOCATE(0);
```

Listing 5.2 – Example code of nested locks and allocation prevention by the HWRM.

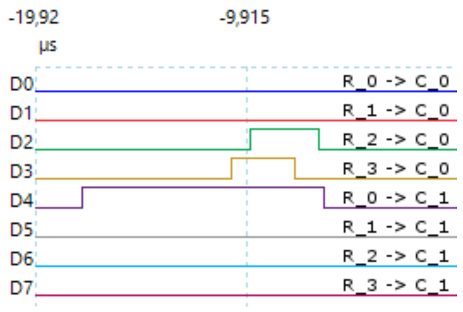


Figure 5.4. – Resource assignment output of the nested locks example in Listings 5.2.

Timeout operation

In a real-time environment the timeout operation is a useful approach to define hard real-time constraints. The HWRM has no knowledge about time. Therefore, this problem has to be handled by the operating system. To manage timeouts, the HWRM offers the timeout command to signal the HWRM to remove the request from the global waiting array. This operation could only be used if the local software waiting queue is empty, otherwise the overwrite command must be used. Listing 5.3 shows an example of the execution of the timeout operation and in Figure 5.5 the resulted resource assignments are shown. Firstly, all cores are requesting for the resource concurrently and the core c_3 gets the resource, because of the highest request priority. Afterwards, it releases the resource and the resource is handed over to core c_2, because it has the highest priority in the waiting array. Here, the interrupts are disabled and therefore, the resource assignment is not signaled to the core. The core c_2 has no knowledge about the assigned resource and executes the timeout operation. The item in the waiting array was already removed because the resource is assigned to the core. Now, the HWRM detects this, releases the allocated resource and hands it over to the next core c_1, which has the highest priority in the array.

```
/* initialization */
SET_CEIL(0,13);

/*C_0*/          /*C_1*/          /*C_2*/          /*C_3*/
ALLOCATE(0, 10); ALLOCATE(0, 11); ALLOCATE(0, 12); ALLOCATE(0, 13);
                                                   DEALLOCATE(0);
               /* C_2 gets the resource */
                                  TIME_OUT(0);
               /* C_1 gets the resource */
```

Listing 5.3 – Example code that illustrate the timeout operation.

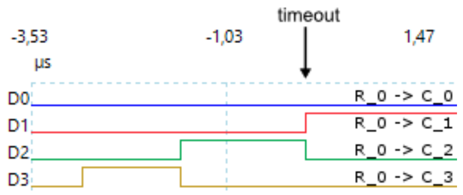


Figure 5.5. – Resource assignment output of the timeout example in Section 5.2.

Overwrite operation

The resource manager offers for every resource and for every core only one space to save a priority of a requesting task. Therefore, in some situations, the array item has to be updated, because of a new head in the software resource waiting queue. The resource manager offers the overwrite operation, which overwrites the array item. Listing 5.4 shows an example of the execution of this operator in line 7. At the beginning of this example, the resource r_0 is requested concurrently from all cores and the resource is assigned to the core with the highest number. In my implementation, the core with the highest number has the highest priority. After releasing the assigned resource from core c_3, it will be assigned to core c_2 because of its highest number. This is also shown in Figure 5.6. The core c_2 that holds the resource r_0 executes an overwrite operation and sets the priority to a lower value than the other resource request priorities inside the array. In this case, the HWRM detects this incorrect assigned resource allocation, releases the resource automatically and assigns it to core c_1. Moreover, the released allocation is inserted into the array to get the resource at a later moment.

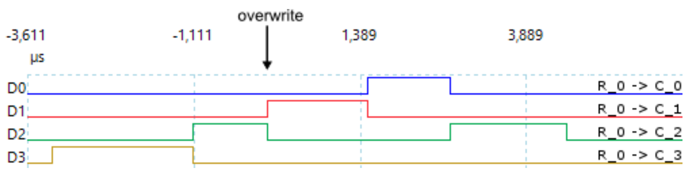


Figure 5.6. – Resource assignment output which shows the recovery handling from the HWRM with the example code in Listing 5.4.

```
/* initialization */
SET_CEIL(0,4);

/*C_0*/          /*C_1*/           /*C_2*/          /*C_3*/
ALLOCATE(0,4);   ALLOCATE(0,4);    ALLOCATE(0,4);   ALLOCATE(0,4);
                                                    DEALLOCATE(1);
                                   OVERWRITE(0,3);
                 SCTL |= EOI;
                 DEALLOCATE(0);
SCTL |= EOI;
DEALLOCATE(0);
                                   SCTL |= EOI;
                                   DEALLOCATE(0);
```

Listing 5.4 – Example code which recover the assignment of a resource by using the overwrite operation.

Interrupt throwing

The aim of a resource management protocol is to reduce the time of priority inversions. The solution in the HMPCP is to boost the priority of the task that causes a priority inversion. The core c_1 with the two assigned resources, in Listing 5.5, causes a priority inversion with both resources. Thus, the priority of the tasks should be increased to their boost priority. The timeout could have the side effect that the priority inversion is eliminated. In that case, the boosted priority should be set back to the task's base priority. The information to let the operating system know about rising or falling of the task's priority is signaled with an interrupt from the HWRM as shown in Figure 5.7. For every assigned resource to a core, an interrupt is generated and the operating system manages, with the information from the HWRM, the priorities of the tasks. Besides, the example code shows the detection of a priority inversion when a rise interrupt is thrown. Firstly, it is detected when the core c_1 makes a request for the just assigned resource r_0, because of its higher priority. Secondly, it detects a priority inversion at the request from core c_2 to the resource r_2, which is not assigned to the core c_0, although it has a higher priority than the assigned resources to core c_0 (1 and 2). This happens, because the priority of core c_2 resource request(4) is not higher than the actual ceiling priority (4).
A rise interrupt is thrown from the HWRM if a task causes a priority inversion. The fall interrupt

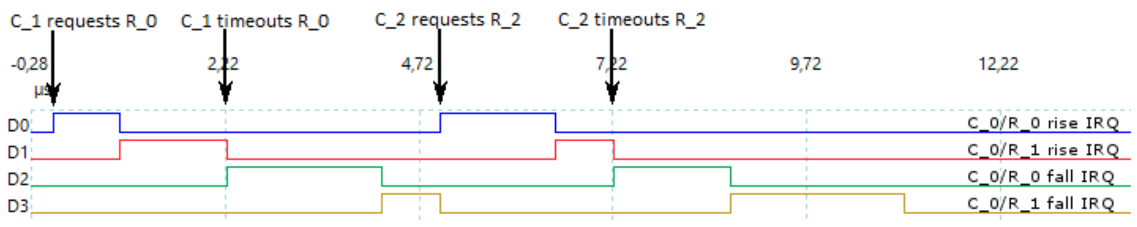


Figure 5.7. – Rise and fall interrupt signals of the example code in Listing 5.5.

```
/* initialization */
SET_CEIL(0, 4);
SET_CEIL(1, 4);
SET_CEIL(2, 4);

/*C_0*/                             /*C_1*/          /*C_2*/
ALLOCATE(0,1);
ALLOCATE(1,2);
                                    ALLOCATE(0,3);
SCTL |= EOI; /*ack rise irq res0*/
SCTL |= EOI; /*ack rise irq res1*/
                                    TIME_OUT(0);
SCTL |= EOI; /*ack fall irq res0*/
SCTL |= EOI; /*ack fall irq res1*/
                                                     ALLOCATE(2,4);
SCTL |= EOI; /*ack rise irq res0*/
SCTL |= EOI; /*ack rise irq res1*/
                                                     TIME_OUT(2);
SCTL |= EOI; /*ack fall irq res0*/
SCTL |= EOI; /*ack fall irq res1*/

DEALLOCATE(0);
DEALLOCATE(1);
```

Listing 5.5 – Example code which produces rise and fall interrupts, by requests from a owner resource and a request from a non owner resource.

is thrown if the priority inversion is eliminated by a timeout of the requested task. The interrupts can be handled only sequentially. Firstly the rise interrupt is thrown because its priority is higher than the one of the fall interrupt. The interrupt could be handled only in situations where the processor is outside kernel mode, because kernel mode deactivates interrupts. Listing 5.6 simulates a situation in kernel mode. Core c_0, which owns the resource has disabled the interrupts. The second core c_1 requests for the resource r_0, but the request is refused because it is not free. An interrupt, to boost the task's priority, is sent to the core c_0 to which the resource is assigned. The interrupt cannot be detected, because of the disabled interrupts. Afterwards the timeout operation is called from core c_1. The fall interrupt will be thrown, after acknowledging the rise interrupt. The request is canceled with the timeout operation and consequently the priority inversion is eliminated. Figure 5.8 demonstrates the output signal of the interrupt signals, the assignment of the resource and the entry of a request in the waiting array. The rise interrupt is immediately thrown after a refused resource request. The queue entry signal is risen after a cycle, but the HWRM takes the refused resource request into consideration just at the time of a resource request. Afterwards, the rise interrupt is canceled, while the queued core is removed by a timeout operation and the fall interrupt is also canceled. With this recovery, the code must not handle the interrupts of a just removed priority inversion.

```
/* initialization */
SET_CEIL(0, 11);

/*C_0*/                               /*C_1*/
ALLOCATE(0, 10);
/* enter kernel mode */
                                      ALLOCATE(0, 11);
                                      TIME_OUT(0);
DEALLOCATE(0);
/* exit kernel mode */
```

Listing 5.6 – Example code where the interrupt is canceled by the HWRM.

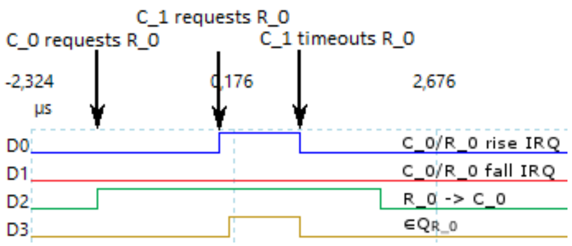


Figure 5.8. – Output signals of the code in Listing 5.6.

5.3. Use case: Shared Resources

This section presents a common resource sharing problem on an embedded real-time system. Suppose that three tasks are running on three different cores $c_0 \dots c_2$. Every task has a different resource access pattern, as shown in Figure 5.9. The task $t_{0,51}$ on the core c_0, with base priority $P_{t_{0,51}} = 51$, uses the GPIO component in a short-term allocation way. In the same way the task $t_{2,52}$ with base priority $P_{t_{0,52}} = 52$ on core c_2 uses the UART component. The third task $t_{1,50}$ on core c_1 uses both resources, the GPIO and the UART, in a long-term way. This means that this task suspends itself while it owns the resources. Task $t_{1,50}$ is the task on the system with the lowest priority, therefore, it could cause the priority inversion problem. The two tasks

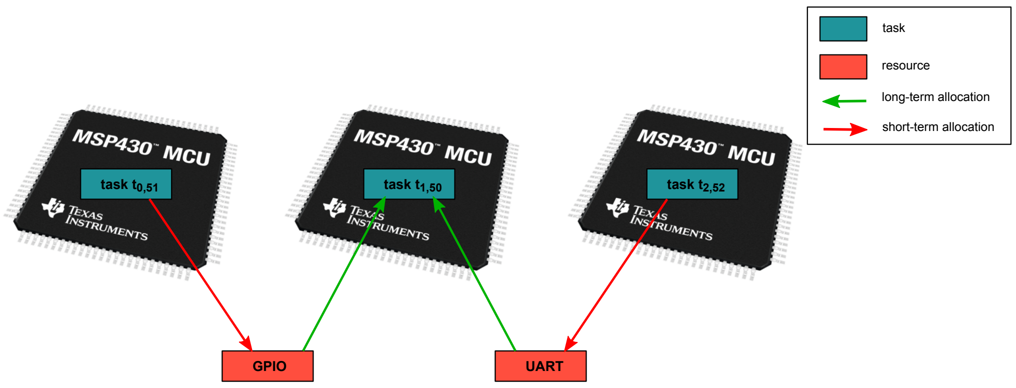


Figure 5.9. – Structure of the shared resources example.

which make a short-allocation, use the resources very shortly: they request the resource, send a header. Then they use the component and, before the resource is released, the tail is sent to close the resource. That critical section never uses a functionality which suspends the task. However, the task $t_{1,50}$ suspends itself during an allocated resource, like demonstrated in Listing 5.1. All tasks already announced the HWRM about their base priority for every used resource. Afterwards, both resources are assigned by the task $t_{1,50}$ and the header of both components is executed. The task accesses the GPIO and UART components. This is followed by a sleep of 200 µs and finally an access to the GPIO component happens one more time. This execution is repeated five times. Afterwards, the components are closed by calling the tail functions and the resources are released. This flow is repeated every 5 ms (period of the task $t_{1,50}$ is $\tilde{T}_{1,50} = 5\,\text{ms}$). Listing 5.1a does not collaborate in the case of a hint. Listing 5.1b however, releases its resources on every hint but needs some extra code for handling it. For the second resource request in line 10, the code checks for an early wake-up. If an early wake-up is thrown, the just assigned GPIO resource is released and the task requests one more time for it. Thereby, the task shortens the time of priority inversion. After getting the resource, the request for the second resource is started one more time. At line 28 the task sleeps for 200 µs. It could return earlier than 200 µs, in case a hint, caused by a priority inversion, is received. In the example both resources are closed and released by the task. Afterwards, it resumes to allocate the resources like at lines 10 to 14. After assigning both resources once again, it checks for a remaining sleep. Otherwise, it resumes with the normal execution flow.
Here, a concurrent run of all tasks is tested. The two higher prioritized tasks $t_{0,51}$ and $t_{2,52}$ are, for convenience, periodic tasks with a period of $T_{0,51} = 3.5\,\text{ms}$ and $T_{0,55} = 3\,\text{ms}$. Certainly, they could also be sporadic tasks. The test measures the blocking time, both for the conventional approach and the one time with the dynamic hinting approach. The results are listed in Table 5.1. The blocking time $B_{0,51}$ contains the time of priority inversion caused by task $t_{1,50}$.

Blocking time	**No hinting**	**With hinting**
average case $B_{2,52}$	688.0 µs	302.7 µs
worst case $B_{2,52}$	794.6 µs	319.4 µs
average case $B_{0,51}$	856.5 µs	399.7 µs
worst case $B_{0,51}$	1229.0 µs	465.1 µs

Table 5.1. – Average and worst case waiting time measurements.

However, the blocking time $B_{2,52}$ contains additionally the priority inversion delay caused by task $t_{0,51}$. This delay is very short, because it allocates the resource, then it does a GPIO operation and releases the resource immediately. The blocking times show that the blocking time of task $t_{0,51}$ is longer than the time of task $t_{2,52}$. The reason for that is that task's $t_{1,50}$ GPIO resource allocation has a longer critical section time than the UART resource assignment. The table also shows, for this test case, a drastic reduction of the blocking time with the usage

```c
OS_DECLARE_TASK(t_1, 100, 50);
OS_TASKENTRY(t_1) {
  Time_t tdeadline;


  getCurrentTime(&tdeadline);
  while (1) {

    while(!getResource(&reGPIO));
    while(!getResource(&reUART));




    uart_header();
    gpio_header();

    for(int i = 0; i < 5; i++){
      Time_t tNext;

      P3OUT |= 0x80;
      putchar(1);
      P3OUT &= ~0x80;

      getCurrentTime(&tNext);
      tNext += 200;

      sleepUntil(&tNext);














    }
    P3OUT &= ~0x80;
    uart_tail();
    gpio_tail();
    releaseResource(&reGPIO);
    releaseResource(&reUART);


    tdeadline += 5000;
    sleepUntil(&tdeadline);
  }
}
```

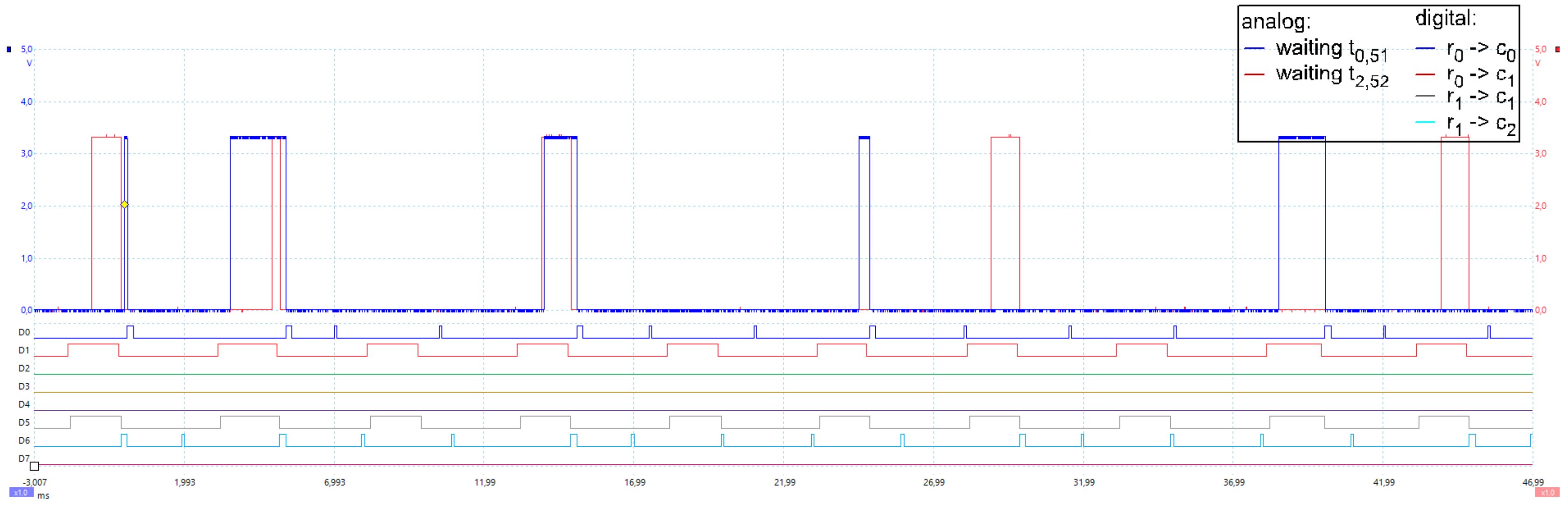
(a) No collaboration on a hint.

```c
OS_DECLARE_TASK(t_1, 100, 50);
OS_TASKENTRY(t_1) {
  Time_t tdeadline;

  os_DHSetEarly(HINT_PRIORITY);
  getCurrentTime(&tdeadline);
  while (1) {

    while(!getResource(&reGPIO));
    while(getResource(&reUART) == -1){
      releaseResource(&reGPIO);
      /* no assigned resource */
      while(!getResource(&reGPIO));
    }
    uart_header();
    gpio_header();

    for(int i = 0; i < 5; i++){
      Time_t tNext;

      P3OUT |= 0x80;      /* -------- */
      putchar(1);         /* workload */
      P3OUT &= ~0x80;     /* -------- */

      getCurrentTime(&tNext);
      tNext += 200;       /* wait 200 us */

      while(sleepUntil(&tNext) == -1){
        /* early wake up */
        uart_tail();
        gpio_tail();
        releaseResource(&reGPIO);
        releaseResource(&reUART);

        /* reallocate both resources */
        while(!getResource(&reGPIO));
        while(getResource(&reUART) == -1){
          releaseResource(&reGPIO);
          /* no assigned resource */
          while(!getResource(&reGPIO));
        }
      }
    }
    P3OUT &= ~0x80;
    uart_tail();
    gpio_tail();
    releaseResource(&reGPIO);
    releaseResource(&reUART);


    tdeadline += 5000;  /* period 5ms */
    sleepUntil(&tdeadline);
  }
}
```

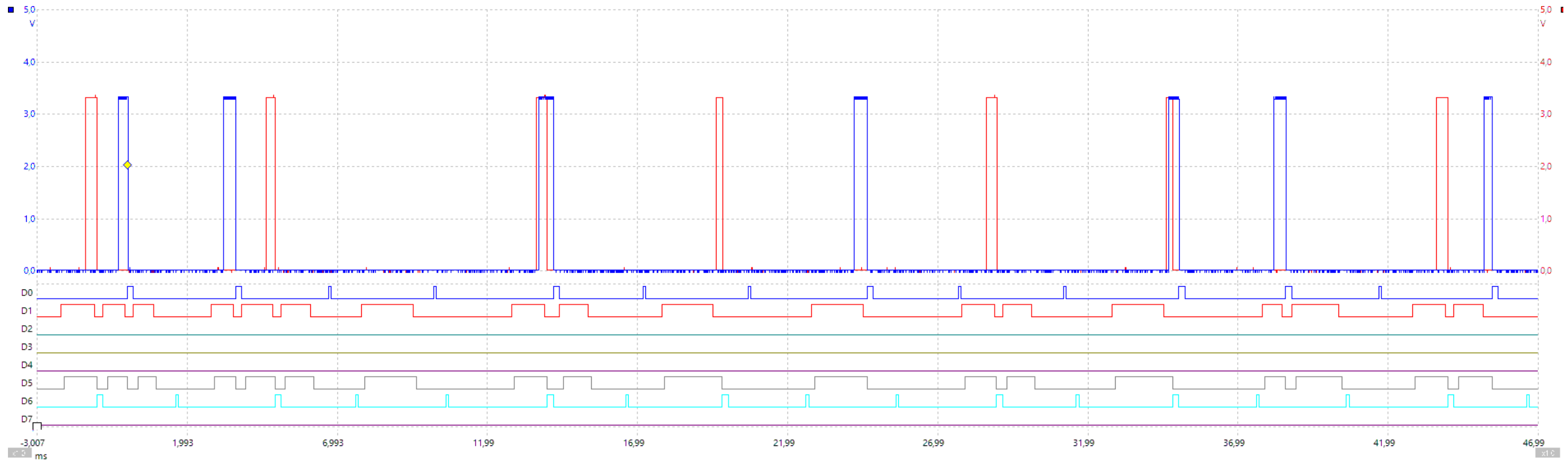
(b) Collaboration on every hint.

Listing 5.1 – Task $t_{1,50}$ which uses two resources (GPIO and UART) in a long-term resource allocation way.

of the dynamic hinting approach. Figure 5.10 graphically shows the blocking time reduction for about the first 50 ms of the simulation. The blue signal in the analog plots is the waiting time of task $t_{0,51}$ and the red one for task $t_{2,52}$. This signals come directly from the waiting array in the HWRM. The digital signals in the bottom of the plots are the resource assignments to the cores. The signal *D0* shows the assignment of the resource r_0 to the core c_0. The second signal (*D1*) shows the assignment of the same resource to the core c_1. Signals *D5* and *D6* demonstrate the assignment of resource r_1 respectively to core c_1 and to core c_2. As the results and the figure show, the approach can reduce the blocking time of higher prioritized tasks, even with higher management overhead.



(a) Output signals without dynamic hinting.



(b) Output signals with dynamic hinting.

Figure 5.10. – Outcomes of the use case example for about the first 50 ms.

6. Conclusion and Future Work

The aim of this thesis was to improve the resource management for real-time multi-core environments. The result is a new resource management protocol, called Hinted Multiprocessor Priority Ceiling Protocol, which is based on the well-known multi-core resource management protocol MPCP and on the dynamic hinting concept. The proposed resource manager is divided into a software and a hardware layer. The HWRM enables cores to access global resources. Moreover, it protects the external peripherals from concurrent access. The HWRM includes the resource management protocol and therefore, it signals the software side about new assignments, priority adjustments, and hints. The hints allow to use the resources in a quasi-preemptive way. Therefore, it can help to reduce the allocation delay of long-time and long-term resources. Upon receiving a hint, a task has the possibility to collaborate, because it knows best how to handle the conflicting resource correctly. The test case has shown that the blocking is extremely reduced in the average-case. Therefore, this new programming paradigm has a high potential and provides many opportunities to improve the real-time resource management for multi-core environments. Nevertheless, there exist many ideas on how to improve the proposed resource management protocol: One extension could be the integration of a hardware timer. This could move the timeout handling from software level to hardware level. The absolute request timeout for a resource is a useful parameter for the information within a hint. The more information the hint contains the better the task could react to it, e.g. by applying a time-utility-function. Another topic could be to analyze the protocol in a more theoretical way, since the schedulability analysis from the initial protocol MPCP does not support the specification of resource allocation timeouts. Additionally, the hinting processing should be integrated into the schedulability analysis. Otherwise, the analysis of the base protocol would probably be too pessimistic. In this work, only global resources are handled. A future work could analyze the integration of local resources into this concept, like nested locks of global and local resources. The implemented HWRM is a centralized component, whose complexity grows with increasing number of cores and resources. As shown in the performance measurements, temporal constraints with more cores or resources might thus not be fulfilled anymore. To eliminate this problem, one could develop a decentralized approach of the proposed concept with constant or even predictable behavior.

Bibliography

[1] T. P. Baker. “A stack-based resource allocation policy for realtime processes.” In: *Proc. 11th Real-Time Systems Symposium*. Vol. 11. IEEE, Dec. 1990, pp. 191–200.

[2] M. Baunach. “Advances in Distributed Real-Time SensoSensor/Act System Operation.” PhD thesis. Julius-Maximilians-Universität Würzburg, 2012.

[3] M. Baunach. “Collaborative Memory Management for Reactive Sensor/Actor Systems.” In: *IEEE 35th Conference on Local Computer Networks (LCN)*. Vol. 35. IEEE, Oct. 2011, pp. 953–960.

[4] M. Baunach. “Dynamic hinting: Collaborative real-time resource management for reactive embedded systems.” In: *Journal of Systems Architecture* 57 (2011), pp. 799–814.

[5] M. Baunach. “Dynamic Hinting: Real-Time Resource Management in Wireless Sensor/Act or Networks.” In: *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA’09)*. Vol. 15. IEEE, Aug. 2009, pp. 31–40.

[6] M. Baunach. “Handling Time and Reactivity for Synchronization and Clock Drift Calculation in Wireless Sensor/Actuator Networks.” In: *Proc. of the 3rd International Conference on Sensor Networks*. 2014, pp. 63–72.

[7] M. Baunach. “Towards Collaborative Resource Sharing under Real-Time Conditions in Multitasking and Multicore Environments.” In: *IEEE 17th Conference on Emerging Technology & Factory Automation (ETFA)*. Vol. 17. IEEE, Sept. 2012, pp. 1–9.

[8] M. Baunach, R. Kolla, and C. Mühlberger. “Introduction to a Small Modular Adept Real-Time Operating System.” In: *6. Fachgespräch Sensornetzwerke*. Aachen, 2007.

[9] A. Block, H. Leontyev, B. B. Brandenburg, and A. James H. “A Flexible Real-Time Locking Protocol for Multiprocessors.” In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. Vol. 13th. IEEE, Aug. 2007, pp. 47–56.

[10] B. B. Brandenburg and J. H. Anderson. “Optimality Results for Multiprocessor Real-Time Locking.” In: *Proc. 31st Real-Time Systems Symposium (RTSS)*. IEEE, Nov. 2010, pp. 49–60.

[11] G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C. Scordino. “Resource Management on Multicore Systems: The ACTORS Approach.” In: *IEEE Micro* 31 (2011), pp. 72–81.

[12] E. Dilger. “Quasi-Präemptive Ressourcenverwaltung in Hardware.” Report at University of Würzburg. 2011.

[13] J. Fornaeus. “Device hypervisors.” In: *Proc. 47th ACM/IEEE Design Automation Conference (DAC)*. Anaheim, CA, USA: IEEE, June 2010, pp. 114–119.

[14] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. “A comparision of MPCP and MSRP when sharing resources in Janus multiple-processor on a chip platform.” In: *9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Vol. 9. IEEE, May 2003, pp. 189–198.

[15] O. Girard. *openMSP430*. 1.14. Dec. 2013.

[16] G. Heiser. “The role of virtualization in embedded systems.” In: *Proc. IIES ’08 Proceedings of the 1st workshop on Isolation and integration in embedded systems*. 2008, pp. 11–16.

[17] *KeyStone Architecture Semaphore2 Hardware Module (SPRUGS3A)*. Texas Instruments. Apr. 2012.

[18] J. Krücken. *How to Configure and Use the XGATE on S12X Devices (AN2685)*. Freescale Semiconductor, Inc. Mar. 2004.

[19] J. Lehoczky, L. Sha, and Y. Ding. “The rate monotonic scheduling algorithm: exact characterization and average case behavior.” In: *Proc. Real Time Systems Symposium*. IEEE, Dec. 1989, pp. 166–171.

[20] F. Nemati, M. Behnam, and T. Nolte. “Independently-developed Real-Time Systems on Multi-cores with Shared Resources.” In: *23rd Euromicro Conference on Real-Time Systems (ECRTS)*. Vol. 23. IEEE, July 2011, pp. 251–261.

[21] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and S. E. Smith. “Multicore Resource Management.” In: *IEEE Micro* 28 (2008), pp. 6–16.

[22] R. Rajkumar, L. Sha, and J. P. Lehoczky. “Real-time Synchronization Protocols for Multiprocessors.” In: *Proc. Real-Time Systems Symposium*. IEEE, Dec. 1988, pp. 259–269.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” In: *IEEE Transations on Computer* 39.9 (Sept. 1990), pp. 1175–1185.

[24] M. Sim. *A Practical Approach to Hardware Semaphores (AN4805)*. Freescale Semiconductor, Inc. Jan. 2014.

[25] *TMS320TVI6487/8 Semaphore (SPRUEF6C)*. 8 February Revised. Texas Instruments. Dec. 2008.

List of Figures

List of Tables

List of Listings

List of Acronyms

ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CPU	Central Processing Unit
C-EDF	Clustered Deadline First
C-SP	Clustered Static Priority
DSP	Digital Signal Processor
EDF	Earliest Deadline First
ELF	Executable and Linking Format
FF	Flip-Flop
FIFO	First In First Out
FMLP	Flexible Multiprocessor Locking Protocol
FPGA	Field Programmable Gate Array
G-EDF	Global Earliest Deadline First
G-SP	Global Static Priority
HMPCP	Hinted Multiprocessor Priority Ceiling Protocol
HWRM	Hardware Resource Manager
GCC	GNU Compiler Collection
GPIO	General Purpose Input/Output
I^2C	Inter-Integrated Circuit
ISR	Interrupt Service Routine
LUT	LookUp Table
MCLK	Main CLocK
MCU	Micro Controller Unit
MMU	Memory Management Unit
MPCP	Multiprocessor Priority Ceiling Protocol
MPU	Memory Protection Unit
MSOS	Multiprocessor Synchronization protocol for real-time Open Systems
MSP430	Mixed Signal Processor 430
MSRP	Multiprocessor Stack Resource Policy
NMI	Non-Maskable Interrupt
OMLP	$\mathcal{O}(m)$ Locking Protocol
OS	Operating System
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol

P-EDF	Partitioned Earliest Deadline First
P-SP	Partitioned Static Priority
RCB	Resource Control Block
RISC	Reduced Instruction Set Computer
SCL	Semaphore Ceiling Register
SCTL	Semaphore Control Register
SFGH	Semaphore Flag High Register
SFGL	Semaphore Flag Low Register
SFR	Special Function Register
SINFH	Semaphore Information High Register
SINFL	Semaphore Information Low Register
SoC	System on Chip
SOP	Semaphore Operation Register
SP	Static Priority
SPI	Serial Peripheral Interface
SRP	Stack Resource Policy
SWRM	Software Resource Manager
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
WCET	Worst Case Execution Time
WNS	Worst Negative Slack
WSAN	Wireless Sensor/Actuator Network
WSN	Wireless Sensor Network

Appendix

A. Resource Manager Registers

Register	Short Form	Register Type	Address	Initial State
Semaphore control	SCTL	Read/Write	0100_H	Reset with POR[5]
Semaphore flags low	SFGL	Read only	0102_H	Reset with POR
Semaphore flags high	SFGH	Read only	0104_H	Reset with POR
Semaphore operation	SOP	Read/Write	0106_H	Reset with POR
Semaphore information low	SINFL	Read only	0108_H	Reset with POR
Semaphore information high	SINFH	Read only	$010A_H$	Reset with POR
Semaphore ceiling	SCL	Write only	$010C_H$	Reset with POR

Register Bit Conventions

Key	Bit Accessibility
rw	Read/write
r	Read only
r0	Read as 0
r1	Read as 1
w	Write only
(w)	No register bit implemented; writing a 1 results in a pulse. The register bit is always read as 0.
-0,-1	Condition after PUC
-(0),-(1)	Condition after POR

[5]Power On Reset

SCTL, Semaphore control

15	14	13	12	11	10	9	8
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	**CF**	**EOI**	**IE**	**IFG**
r0	r0	r0	r0	rw-(0)	r0(w)	rw-(0)	rw-(0)

CF Bit 3 Ceiling finish. Setting CF = 1 notifies the resource manager that the core has announced all the used resources with the associated task priority. The flag CF will be read with 1 if all cores have set the CF flag.

EOI Bit 2 End of interrupt. Setting EOI = 1 clears the interrupt flag and notifies that the interrupt was handled.

1 The interrupt is handled and is ready for the next interrupt.

IE Bit 1 Interrupt enable. This bit enables the IFG interrupt request.

0 Interrupt disabled.

1 Interrupt enabled.

IFG Bit 0 Interrupt flag. Is automatically cleard if the EOI is setted.

0 No interrupt pending.

1 Interrupt pending.

SFGL, Semaphore flags low

15	14	13	12	11	10	9	8
SFG15	**SFG14**	**SFG13**	**SFG12**	**SFG11**	**SFG10**	**SFG9**	**SFG8**
r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)

7	6	5	4	3	2	1	0
SFG7	**SFG6**	**SFG5**	**SFG4**	**SFG3**	**SFG2**	**SFG1**	**SFG0**
r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)

SFG	Bit n	Semaphore flag.	
		0	Semaphore n is assigned to the core.
		1	Semaphore n is not assigned to the core.

SFGH, Semaphore flags high

15	14	13	12	11	10	9	8
SFG31	**SFG30**	**SFG29**	**SFG28**	**SFG27**	**SFG26**	**SFG25**	**SFG24**
r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)

7	6	5	4	3	2	1	0
SFG23	**SFG22**	**SFG21**	**SFG20**	**SFG19**	**SFG18**	**SFG17**	**SFG16**
r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)

SFG	Bit n	Semaphore flag.	
		0	Semaphore n+16 is assigned to the core.
		1	Semaphore n+16 is not assigned to the core.

SOP, Semaphore operation

15	14	13	12	11	10	9	8
	OP		**SNUM**				
r0	w	w	w	w	w	w	w

7	6	5	4	3	2	1	0
STATUS	**PRTY**						
r	w	w	w	w	w	w	w

OP	Bits 14-13	Operation. Defines the operation which is sent to the resource manager.	
		00	Release semaphore. Deallocates the actually assigned resource. The command is ignored if the resource is not assigned to the core.
		01	Allocate semaphore. Makes a resource request and returns a 1 in the STATUS field if the allocation was successful.
		10	Timeout. Remove the queued request from the waiting array.
		11	Overwrite. Overwrite the array item with a new value. A 1 is returned in the STATUS field if the resource was already assigned to the core.
SNUM	Bits 12-8	Semaphore number. Defines the semaphore number to which the operation refers to.	
STATUS	Bit 1	Operation status. Shows the return value of the allocation/overwrite operation.	
		0	Semaphore is not allocated to the core.
		1	Semaphore is allocated to the core.
PRTY	Bits 6-0	Priority. The priority of the task which calls the operation.	

SINFL, Semaphore information low

15	14	13	12	11	10	9	8
0	**RPRTY**						
r0	r	r	r	r	r	r	r

7	6	5	4	3	2	1	0
AF	**RF**	**FF**	**SNUM**				
r	r	r	r	r	r	r	r

RPRTY	Bits 14-8	Requester-priority. Shows the priority of the blocked task in case of a hint.
AF	Bit 7	Allocation flag. 1 The interrupt is an allocation notification.
RF	Bit 6	Rise and hint flag. 1 The interrupt is a rise priority/hint notification.
FF	Bit 5	Fall flag. 1 The interrupt is a fall priority notification.
SNUM	Bits 4-0	Semaphore number. The interrupt is associated with this semaphore number.

SINFH, Semaphore information high

15	14	13	12	11	10	9	8
0	**HCEIL**						
r0	r	r	r	r	r	r	r

7	6	5	4	3	2	1	0
0	**CEIL**						
r0	r	r	r	r	r	r	r

HCEIL	Bits 14-8	Highest Ceiling. The ceiling priority of all resources on the entire system.
CEIL	Bits 6-0	Ceiling. The ceiling priority of the semaphore number SNUM in SINFL.

SCL, Semaphore ceiling

15	14	13	12	11	10	9	8
0	**0**	**0**	**CSNUM**				
r0	r0	r0	w	w	w	w	w

7	6	5	4	3	2	1	0
0	**PRTY**						
r0	w	w	w	w	w	w	w

CSNUM	Bits 12-8	Ceiling Semaphore number. The ceiling priority is set for this semaphore number.
PRTY	Bits 6-0	Priority. The priority of the task that uses the semaphore number CSNUM in this register.

B. SmartOS extended data structure

Resource Control Block (RCB)

```
typedef struct {
  Event_t   release_event ;        // the event connected to this resource
  TaskId_t  owner ;                // the owner task ( NIL_ID for not allocated )
  char      flags ;                // flags for Dynamic Hinting
  char      *name ;                // resource name ( for debugging only )
  short     (* fInit )( void );    // resource initialization
  short     (* fGet )( const Time_t * deadline ); // called during first allocation
  short     (* fRelease )( void ); // called during last deallocation
  short     globalID;              // global resource ID ( 0xFFFF = local resource )
} Resource_t ;  /* size = 7 words */
```

C. SmartOS extended API

Resources Declaration

```
OS_DECLARE_RESOURCE(name);
OS_DECLARE_RESOURCE_EXT(name, f *fInit, f *fGet, f *fRelease);
OS_DECLARE_GLOBAL_RESOURCE(name, id);
OS_DECLARE_GLOBAL_RESOURCE_EXT(name, id, f *fInit, f *fGet, f *fRelease);
OS_IMPORT_RESOURCE(name);
```

Resources Functions

```
void releaseResource(Resource_t *resource);
int getResourceUntil(Resource_t *resource, Time_t *deadline);
int getResourceFor(Resource_t *resource, Delay_t timeout);
int getResource(Resource_t *resource);
int testResource(Resource_t *resource);
int isResourceOwner(TaskId_t taskID, Resource_t *resource);
```