Design of a Lightweight Real-Time Streaming Kernel

Bas van Sisseren

Distributed and Embedded Systems,
University of Twente.

Date:
August 13, 2007

Committee:
ir. P.G. Jansen
prof. dr. ir. G.J.M. Smit
ir. M.H. Wiggers
ir. T. Hofmeijer

Abstract

This report describes a flexible real-time kernel, which is optimised for data-streams, to be used for multi-processor environments.

Currently, two processors are used: the MSP430 and the ARM 946E-S. For the first architecture, an in-house kernel has been developed by Tjerk Hofmeijer. For the ARM architecture, the currently available kernel implementations either lack support for dynamic real-time scheduling or are not available.

This document describes the kernel *BasOS*, which was developed within this project. BasOS is a flexible real-time kernel with low memory usage, efficient interrupt handling, both real-time and non-real-time scheduling. The kernel has a programmer-friendly interface and supports several peripherals, like the USART (serial port), USB and the Montium processors.

Also, several tools, which support the use of BasOS: a stack usage predictor, a loader of dynamic tasks and a second stage boot loader.

Preface

In this report, I have described the aspects of my assignment to extend and implement a real-time kernel for the Basic Concept Verification Platform (BCVP) and Highly integrated Concept Verification Platform (HiCVP). Unfortunately, the HiCVP is still under development. Therefore all development has been done on the BCVP, the predecessor of the HiCVP.

This document is mainly written for people interested in the kernel mechanisms of BasOS and for those who would like to write device drivers and applications for the kernel.

The reasons why I have chosen to implement a kernel is because I've always wanted to write a system without being dependent on other software. The other reason is that I like to bind optimal solutions to complex environments.

I hereby would like to thank my committee for supporting me in various ways in the process of designing the kernel and writing this report. I would also like to thank Pascal Wolkotte and Lodewijk Smit for helping me find my way on the BCVP platform and the extensive support they have given me. I would like to thank Albert Molderink and Marcel Hamer for putting various parts of the kernel into discussion. Last but not least, I would like to thank the many proofreaders of this document.

Contents

Chapter 1

Introduction

A kernel is the basis for every operating system of a computing platform. It is a program that acts as an intermediary between between a user of a computer and the computer hardware. It supports basic functions such as memory management, hardware interrupt handling and task-switching. Furthermore, it gives an application an environment in which it can run, has support for inter-task communications, divides resources between the running applications and handles the communication with hardware.

Compared to generic kernels (e.g. in Windows, Linux), real-time kernels not only just divide the available resources between the running applications, but can also guarantee that the application will run with timely guarantees, under the assumption that the application respects its real-time constraints.

In a real-time system, an application is split in one or more tasks. Each task can have a set of runtime constraints, such as task duration, task deadline and task resource usage.

There are two most common types of real-time systems. Hard real-time systems, where missing such a deadline can result in disaster, and soft real-time systems, where missing a deadline only results in a performance loss.

1.1 Problem description

Within the Embedded Systems group at the University of Twente, there is a need for such a soft real-time kernel. Most projects currently use the Dimitri or eCos kernels. These kernels offer too restricted scheduling possibilities or have a large memory footprint (more than 100 KB) and unnecessary overhead due to their supported features like a runtime reconfigurable hardware abstraction layer with excessive use of callback functions and in-kernel debugging.

The aim for this project is to have a lightweight real-time kernel for the platform that is currently used most at the Embedded Systems group, that is the Basic Concept Verification Platform (BCVP). The BCVP board has two ARM9 processors, several timers, a serial port, an USB device-port and several more peripherals. The BCVP also has an FPGA board, which can be used for emulating a Montium processor [6].

The kernel we are designing will be used for streaming real-time applications, such as MPEG4 decoding, and test-applications for the Montium environment. The kernel needs to be able to control the serial port, the USB device-port and the Montium processor, while keeping a low memory footprint (less than 50 KB). Features which this kernel should have are memory allocation, advanced interrupt handling, flexible real-time Quality of Service (QoS) scheduling (preferably EDFI) and dynamic task insertion and deletion.

1.2 Document overview

Chapter 2 will describe existing work. Current kernel implementations will be discussed. An introduction in the BCVP architecture is given. Furthermore, fast interrupt handling by using impulse handlers and the EDFI scheduling algorithm is described.

In Chapter 3, the design of the kernel itself will be described. What does the application interface, that we have in mind, look like. Which adaptations do we have to make to the given theories and which problems do we expect to have.

Chapter 4 describes implementation issues. What are the trade-offs between the possible choices. Why did we choose for a specific implementation.

Chapter 5 gives an overview of the currently available device drivers in BasOS. It also describes how these devices can be accessed from an application.

In Chapter 6, the interaction with the kernel is described. What should an application implement. How is a dynamic task specified and how is it activated within a running kernel.

Chapter 7 will give a list of all recommendations.

Finally, the conclusions can be found in Chapter 8.

Chapter 2

State of the Art

This chapter gives an overview of previous work. It describes existing kernel implementations, the Basic Concept Verification Platform, scheduling techniques and interrupt handling.

2.1 Current implementations

There are many existing real-time and non-real-time kernels available. As stated in Chapter 1, every kernel has its own characteristics. Most kernels are not written with few memory in mind and often do not have support for real-time tasks. Others are, but are often optimised for one or several pre-defined task-sets.

Within the 4S-project [11] a long list of kernels have been evaluated. From this list, the most interesting kernels have been selected. The criteria we used for selecting these kernels were:

- Is the source available.
- Does the kernel have support for the ARM architecture.
- Are there drivers available for the BCVP peripherals.
- Does it have a low memory footprint. (less than 50 KB)
- Does it have support for real-time scheduling.

None of the selected kernels matched all criteria, but four kernels were close enough. We will have a closer look at these four kernels.

2.1.1 DCOS

DCOS [4] is a lightweight kernel, written for the MSP430 processor. It has been developed by Tjerk Hofmeijer at the University of Twente. The kernel uses EDFI scheduling [5] and first-fit memory heap allocation. It has a low memory footprint. It uses impulses [7] for fast interrupt handling. Unfortunately, the only implementation available is for the 16-bit MSP430 and it only has support for the EDFI scheduling algorithm. (see Section 2.4.1 for a description of EDFI scheduling)

2.1.2 eCos

eCos [14] is a rather complete kernel. It is developed within the open source community and there is an implementation available for the BCVP platform. The kernel has USB, ethernet, serial and flash support. Unfortunately, the kernel does not have support for real-time tasks. Also, the kernel is not very memory efficient.

2.1.3 TinyOS

TinyOS [17] is a very lightweight kernel developed for wireless embedded sensor networks. It was initially developed by the U.C. Berkeley EECS Department. Currently, numerous groups are actively contributing code to the project. TinyOS uses a pre-compiled set of tasks, which makes it less suitable for dynamic task loading and task migrations. There is no ARM support for TinyOS.

2.1.4 RTlinux

RTlinux is one of the more familiar real-time kernels. It is built on top of (or actually below) Linux. Therefore, when using this kernel, running Linux is also necessary, which needs many resources. An advantage is that interaction with Linux facilitates writing tasks for this platform. Due to its large footprint, this kernel is less interesting than the previously discussed kernels.

2.1.5 Summary

None of the described kernels is flexible enough to suit as a basis for our kernel. Adapting one of these already existing kernels would take more time than partially redesigning and re-implementing it. When we would choose to adapt such a kernel, this would lead to matching ideas that are not well suited to live together.

Instead, we prefer to re-use only parts of the code of the existing kernels. For example, the EDFI scheduling and heap management from the DCOS kernel and the eCos BCVP driver implementations are interesting enough to be used in our own kernel.

2.2 The Basic Concept Verification Platform

We are using the BCVP (seen in Figure 2.1) as the development platform for our kernel. In the 4S-project [11], it is used for building applications in the field of Digital Radio Mondiale (DRM) and MPEG4 decoding. The 4S-project mission statement is:

> 4S will realize flexible and reconfigurable building blocks to pave the way for new consumer devices and applications like digital information broadcasting, ambient intelligence devices and 3G/4G multimedia terminals.

Eventually, the BCVP will be replaced by the *Highly integrated Concept Verification Platform* (HiCVP), but since the HiCVP is not available yet, the BCVP is the best alternative. There are only a few small changes between the BCVP and HiCVP. On the BCVP there are two ARM processors available. On the HiCVP there is just one ARM processor. Furthermore, peripherals use different memory addresses and interrupt vectors.

Figure 2.1: The Basic Concept Verification Platform (BCVP)

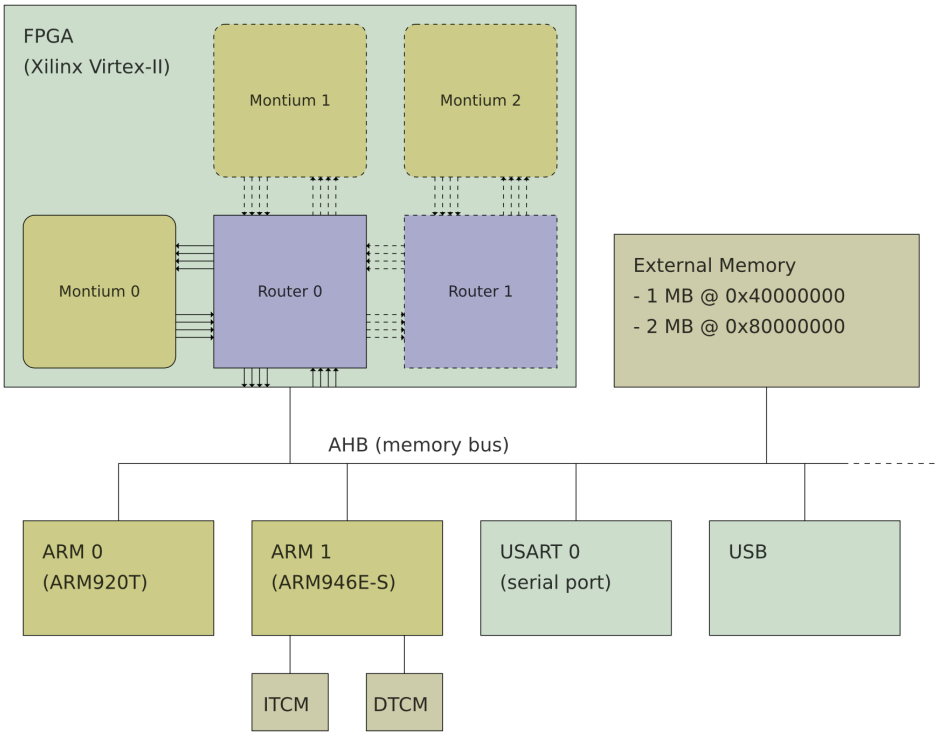

Figure 2.2: Schematic overview of the BCVP

A simplified schematic overview of the BCVP is given in Figure 2.2. The BCVP has two ARM processors, the ARM920T (on the BCVP known as ARM0) and the ARM946E-S (on the BCVP known as ARM1). Both processors can access all peripherals by using the memory bus. Only the currently used peripherals are shown. We will give a description of all given blocks.

ARM0
The ARM0 (ARM920T) processor is one of the two processors available on the BCVP. On boot, the ARM0 processor is disabled.

ARM1
The ARM1 (ARM946E-S) processor is the second processor available. On boot, this processor starts the *RedBoot* application, found in flash memory. RedBoot [15] provides a simple command-line interface for loading other applications.

ITCM
Instruction Tightly-coupled Memory. The ITCM is only accessible from ARM1 and has a size of 32 KB.

DTCM
Data Tightly-coupled Memory. The DTCM is only accessible from ARM1 and has a size of 64 KB.

External memory
The BCVP has 3 MB of external memory. One block of 1 MB is available for ARM0, but can also be accessed by ARM1 and one block of 2 MB is available for ARM1 (ARM0 cannot access this memory).

USART0
Universal Synchronous Asynchronous Receiver Transmitter. In total, three USARTS are available. The BCVP has one serial port available, which can be controlled via USART0. (the other two USARTs are used for communications inside the BCVP board)

USB
USB Device Peripheral. This peripheral acts as a USB device on the USB bus.

FPGA
There are two variants of the BCVP. One has an FPGA Virtex-II 3000 and the other an FPGA Virtex-II 8000. An FPGA is configurable hardware; it emulates one or more Montium processors, its Hydra CCU and a routing network. (see the descriptions below)

The Virtex-II 3000 can only emulate one router and one Montium, while the Virtex-II 8000 can emulate two routers and three Montiums.

Routers
The routing network routes all communication between the BCVP and Montiums and between the Montiums itself. The routing network is a connection oriented switch for Montium processors. [6]

Montium
The Montium processor [6] is an energy-efficient, reconfigurable processor, which has been developed at the University of Twente. It is optimised for highly regular computations.

The BCVP emulates the Montium tiles by using an FPGA, the planned HiCVP will have four Montium tiles.

Hydra CCU

The Hydra CCU (Hydra Communication and Control Unit) [6] is the interface which communicates with the Montium processors. It can load new code into the Montium chips, read and write parameter blocks, and start and stop the Montium processor.

The BCVP also has a MultiICE debugging interface available, but this interface caused huge system call overhead and is therefore not used anymore.

2.2.1 The ARM Architecture

On the BCVP, the ARM946E-S (in BCVP documentation often referred to as ARM1) is always online. It can be debugged via the JTAG interface. Therefore development has mainly taken place using this CPU.

Compared to the ARM946E-S, the ARM926EJ-S (the ARM processor that is available on the HiCVP) is quite similar. In addition to the ARM946E-S, it has support for running Java byte-code and has support for virtual memory.

A short introduction in the ARM architecture will now be given. (see the ARM reference documentation [12] for a more complete specification)

ARM CPU-Modes and Exceptions

The ARM architecture has 7 CPU-modes and 16 general-purpose registers, including the program counter register. Some registers are "banked", which means that these registers are masked by a CPU-mode specific register. Every CPU-mode has its own stack pointer. (see Table 2.1 for all registers and CPU-modes)

System / User	**Fast Interrupt (FIQ)**	**Supervisor (SVC)**	**Abort (ABT)**	**Interrupt (IRQ)**	**Undefined (UND)**
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	*r8_fiq*	r8	r8	r8	r8
r9	*r9_fiq*	r9	r9	r9	r9
r10	*r10_fiq*	r10	r10	r10	r10
r11	*r11_fiq*	r11	r11	r11	r11
r12	*r12_fiq*	r12	r12	r12	r12
r13	*r13_fiq*	*r13_svc*	*r13_abt*	*r13_irq*	*r13_und*
r14	*r14_fiq*	*r14_svc*	*r14_abt*	*r14_irq*	*r14_und*
r15	r15	r15	r15	r15	r15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	*SPSR_svc*	*SPSR_abt*	*SPSR_irq*	*SPSR_und*

Table 2.1: Available ARM registers

All CPU-modes except for the user mode are privileged CPU-modes. In these modes, the task can access CPU-specific registers (e.g. for configuring memory protection or changing the CPU-mode).

The ARM architecture also specifies seven exception vectors, normally placed at memory offset 0. These exceptions are used for handling a soft reset, an error or an interrupt. When such an exception is called, the processor switches to the exception's CPU-mode to keep the user's registers intact. In Table 2.2 an overview is given of all exceptions and their CPU-mode.

Address	**Exception**	**CPU-mode**	**Description**
0x00000000	Reset	Supervisor (SVC)	
0x00000004	Undefined instruction	Undefined (UND)	
0x00000008	Software interrupt	Supervisor (SVC)	System calls
0x0000000c	Instruction abort	Abort (ABT)	Read-error on instruction-fetch
0x00000010	Data abort	Abort (ABT)	Read- or write-error
0x00000018	Interrupt	IRQ	
0x0000001c	Fast interrupt	FIQ	

Table 2.2: ARM Architecture Exception Vectors

2.2.2 Memory Layout

The BCVP has a total of 3 MB RAM available. The ARM920T (ARM0) can only access 1 MB, the ARM946E-S (ARM1) can access all memory. All peripherals use memory-mapped IO, as is common on ARM architectures. Most are available above address **0xf0000000**. See Table 2.3 for the BCVP memory layout.

Interesting to note is that the internal memory block contains two Tightly-coupled Memory (TCM) blocks. These blocks can be accessed more quickly than the external memory. One block with a size of 32 KB is optimised for instructions (the ITCM) and the other, which has a size of 64 KB, is optimised for data (the DTCM). If the kernel is small enough, placing the code in TCM will yield an interesting performance gain.

Address	Description
0x00000000	Internal memory
0x30000000	Montium Hydra
0x40000000	1 MB RAM (ARM0)
0x50000000	Flash memory
0x80000000	2 MB RAM (ARM1)
0xf0000000	Peripherals

Table 2.3: BCVP physical memory layout

2.3 Impulse handling

In a single processor environment, it is in general preferred to handle interrupts in an interrupt disabled state. When updating various kernel variables from an interrupt handler, it should be guaranteed this variable cannot be changed by another interrupt handler. On the other hand, interrupts could be left in a disabled state, but this might be a risk for interrupt handlers that need to react quickly.

A way to solve this problem is by using impulse handling [7]. The interrupt handler is then split in two halves. The first half, directly called on interrupt and still in interrupt-disabled state, can acknowledge the interrupt and check whether the second half should be run. If the second half should run, a flag is set.

When the first half has finished processing, it checks whether there are pending interrupts. If there are interrupts waiting and the impulse handler is not already active, the impulse handler is flagged as active, the processor is set in interrupt-enabled state and the delayed second half handlers will be processed with a specific priority. By handling the pending interrupts sequentially, only one handler can be active at a time. A handler can then safely update internal kernel variables without the risk of a second handler reading or updating the same resource.

Since the processor is in interrupt-enabled state while handling the second half handlers, it is possible that a new pending interrupt will arrive. The already running impulse handler will pick this up and the handling works as expected.

After all second half handlers have been handled, the impulse handler is flagged as inactive, the processor is set in interrupt-disabled state and the interrupt handler will return.

The interrupt-handler pseudo-code is as follows:

```
interrupt:
  handle_interrupt_first_half();
  if ( impulse_bits_set && ! impulse_handler_running )
  {
    /* run impulse handler */
    impulse_handler_running = true;
    enable_interrupts();
    while( delayed_interrupt_waiting() )
    {
      handle_delayed_interrupts();
    }
    disable_interrupts();
    impulse_handler_running = false;
  }
  interrupt_return; // enable interrupts, restore CPU-mode and return
```

2.4 Real-time Scheduling

Assume an application with a task-set Ω as given in Table 2.4. There are four tasks (a task has symbol τ_i, with $1 \leq i \leq 4$) and each task has its own relative deadline (D_i), period (T_i), runtime (C_i) and shared resources. Every T_i time units, the task τ_i is released, after which it has D_i time units to complete its task. The release time of a task of invocation j of τ_i is r_i^j. Then the absolute deadline of this release is $r_i^j + D_i = d_i^j$.

The definition of a resource, given by Butazzo [3, page 181]: "A resource is any software structure that can be used by a process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, or a set of registers of a peripheral device."

In our example, there are two resources, A and B. When a task needs exclusive use of a resource, the resource is denoted in uppercase. When a task allows sharing of the resource, the resource is denoted in lowercase. For instance, reading from a memory block can be shared between tasks, but writing to this memory block should be done exclusively by one task at a time.

The inherited deadline Δ_i is the smallest deadline interval of a task with which τ_i shares an exclusive resource. For instance, task 3 shares resource A with task 1 and 2. Both task 1 and 2 need exclusive use of the resource. Task 3 inherits the smallest deadline interval, which is the deadline of task 1.

Ω	D_i	T_i	C_i	Δ_i	Shared resources
τ_1	11	19	2	11	{ A }
τ_2	19	23	5	11	{ A B }
τ_3	25	31	7	11	{ a }
τ_4	30	37	11	19	{ B }

Table 2.4: A task-set Ω

2.4.1 Earliest Deadline First with deadline Inheritance (EDFI)

There are various real-time scheduling algorithms, differing in performance, complexity and applicability. The most common algorithms are compared in [3] on pages 75 and 107. One of the more interesting scheduling algorithms is Earliest Deadline First with deadline Inheritance (EDFI), described in [5]. EDFI is an extension on Earliest Deadline First (EDF), which gives us the possibility to control shared resource locking from within the scheduler itself (this form of real-time tasks is also known as real-time transactions). The EDFI algorithm is, just as the EDF algorithm, lightweight and gives good results on generic tasks.

We give a short introduction on how EDFI scheduling works. A task-set example can be found in Table 2.4:

The EDFI scheduler (see Figure 2.3) has two queues, the wait and released queue, and one run stack, all ordered to absolute deadline (earliest deadline first). Every task is periodically released (every T_i time units) from the wait queue into the released queue.

When a task at the head of the released queue (τ_h) has an earlier deadline than the currently running task (τ_r) and the absolute deadline (D_h) is smaller than the inherited deadline of the currently running task (Δ_r), τ_h will preempt τ_r and become the new running task. (In short, preemption will take place when $d_h < d_r \wedge D_h < \Delta_r$)

When a task finishes or reaches its deadline, the task is removed from the run stack (or released queue) and inserted back into the wait queue, waiting for its next release.

2.4.2 EDFI Feasibility Analysis

For determining whether a given task-set is feasible, we have to examine the *processor demand* $H(t)$, the *workload* $W(t)$ and the *blocking load* $C_B(t)$. $H(t)$ represents the total amount of CPU time that must be available between 0 and t for Ω to be schedulable to make all deadlines met so far. $W(t)$ represents the cumulative amount of CPU time that is consumable by all task releases between time 0 and t. [5, page 3]

$C_B(t)$ is the possible blocking load, caused by the shared resources. L is the point where $W(t) = t$. (the point where the CPU first becomes idle)

The task-set Ω is feasible [5] if

$$\forall t \in \langle 0, L] : H(t) + C_B(t) \leq t.$$

The feasibility analysis of a task-set can be represented in a figure. Figure 2.4 shows an analysis of the task-set given in Table 2.4. The diagonal in the graph represents the amount of work done. The vertical distance between the $W(t)$ and the diagonal represents the amount of work still to do in released tasks. At point L, which is the point where the diagonal touches the $W(t)$ function, there is no more work to do and the system becomes idle. The $H(t)$ function represents the amount of work that must be finished. If $H(t)$ crosses the diagonal, then more work would have to be finished than there is time available. The $C_B(t)$ represents the maximum potential blocking load, which is given by $C_B(t) = max_\Omega\{C_\tau | \Delta_\tau \leq t < D_\tau\}$. The schedulability analysis tracks $W(t)$ and $H(t)$ until either $W(t)$ touches the diagonal or $H(t) + C_B(t)$ crosses it. If $H(t) + C_B(t)$ crosses the diagonal, the task set is not schedulable. If $W(t)$ touches before $H(t) + C_B(t)$ could cross, the task set is feasible.

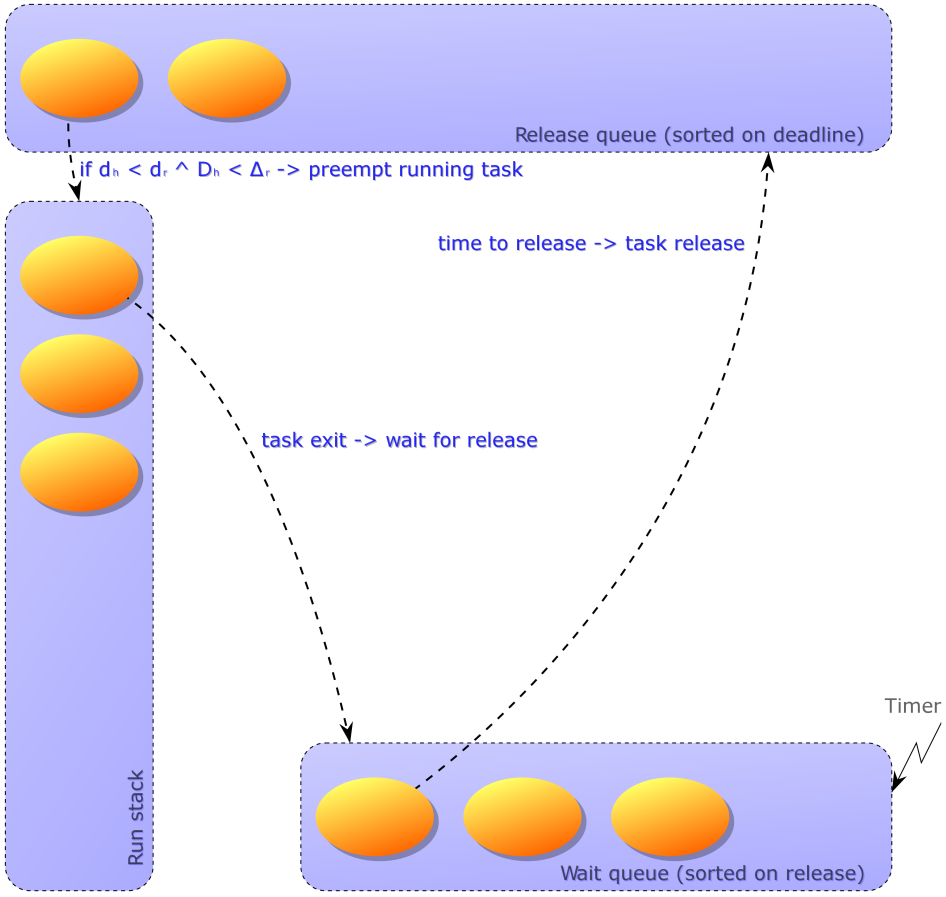

Figure 2.3: EDFI Scheduler: wait queue, release queue and the run stack

Below the graph, the task bars are shown with a possible scheduling. An upward arrow represents the release of the task. A downward arrow represents the deadline of the task. A small downward arrow represents the inherited deadline of the task. The blocks represent a possible scheduling of the given task-set.

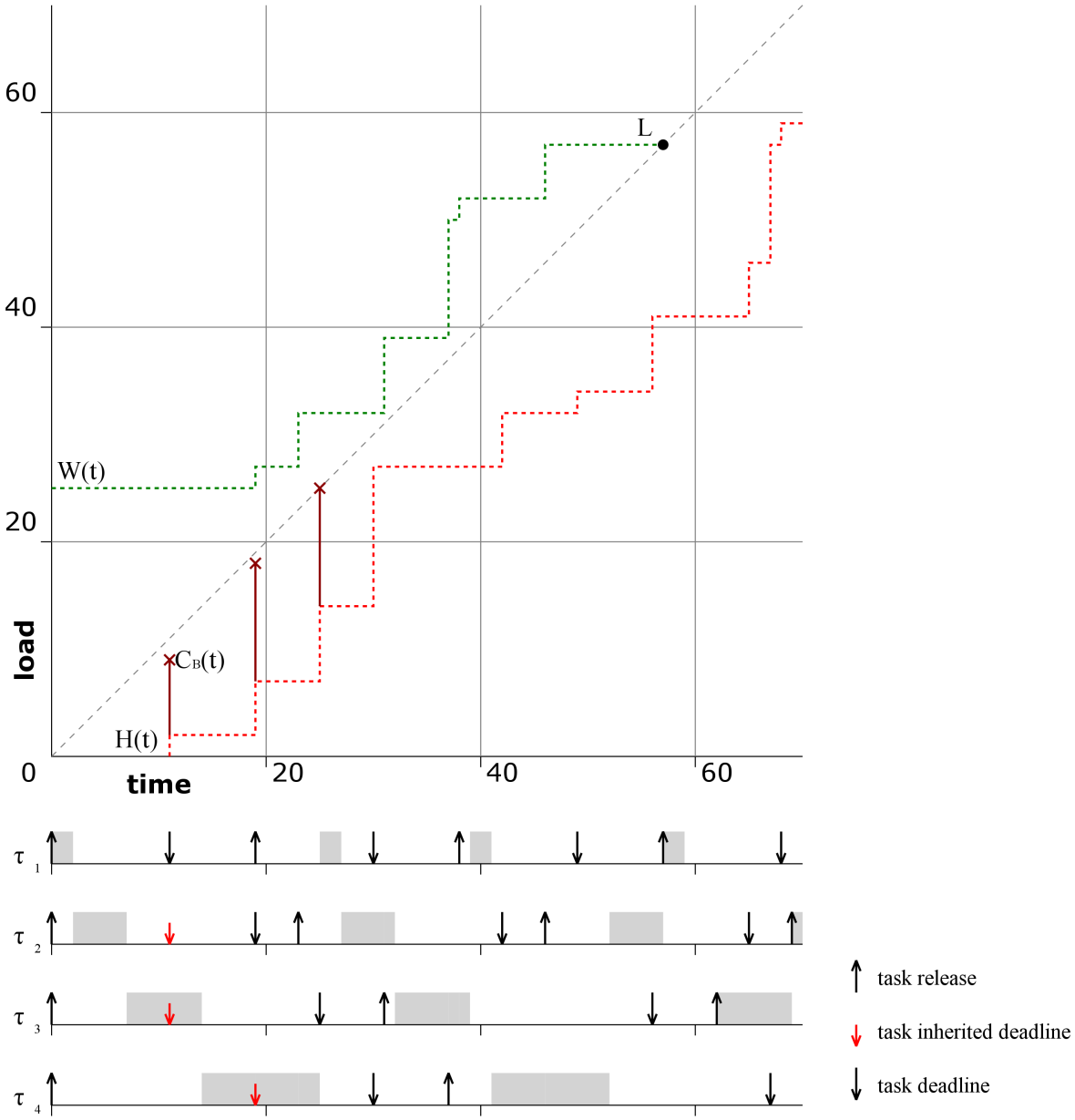

Figure 2.4: EDFI feasibility analysis of the task-set in Table 2.4

Chapter 3

Kernel Design

The emphasis of this chapter is on the design of the kernel. Questions are: what interface should a kernel offer to its applications. Which design decisions have been made.

3.1 Application Interface

A processor should run applications. The interaction with the kernel should be simple and minimal. The kernel is to provide a convenient interface between the computer hardware and the applications. The main tasks of the kernel are scheduling, memory management, interrupt handling and communication with devices.

3.1.1 Tasks

Every application consists of a number of tasks. Every real-time task has its own set of real-time constraints, which we already mentioned in Section 2.4 (relative deadline, used processor time, the period between two consecutive task instances, shared resources, etc.). A task often responds to a certain input, after which it produces an output. Especially in streaming applications, this can result in data-flows as given in Figure 3.1. One task receives input from an external input. When the task is released, it processes this input and sends the result to one or more other tasks.

One of the ideas of BasOS is to optimise this process by partially moving the data streams into the kernel and let the scheduler use this knowledge to minimise delays. These techniques will be described further on in this section.

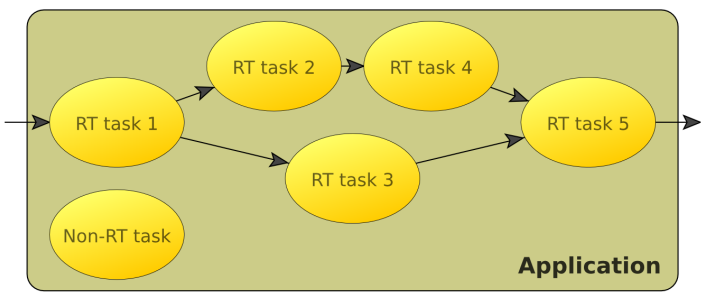

Figure 3.1: Tasks: an application can be seen as a number of tasks

Non real-time tasks

For our target applications, we in general prefer real-time tasks. However, for management and configuration we can also use non-real-time tasks. Reasons for using non-real-time tasks are:

- Non-real-time tasks will not be terminated when they miss their deadline, because there is no deadline.
- In a non-real-time task, we can wait for events to take place. (e.g. wait until data is available in a pipe)
- Using dynamic memory allocation cannot give us real-time guarantees at the moment. If we use dynamic memory allocation in a real-time task, a deadline could result in memory leakage.
- When configuring a new set of real-time tasks, the kernel needs to dynamically allocate new task-structures, pipe-structures, etc. This depends on dynamic memory allocation, which cannot be done real-time.

For these tasks, the kernel uses a simple TDMA scheduling algorithm, which runs in the slack time of the real-time scheduler. With this scheduling model, the scheduler sequentially gives every non-real-time task a time-slice of a predefined amount of time. When a real-time task is activated, the non-real-time task is preempted. On return, the non-real-time scheduler continues with the preempted task, which then runs for the time left over in its current time-slice.

3.1.2 Signals and conditions

In conventional EDFI scheduling a task is periodically released. When dealing with input that is not always immediately available, as in a streaming-oriented environment, it is preferable to activate a task when all conditions for activating are met.

A condition is met when the kernel has received a signal for a specific task. A signal can be received from other tasks, external events or by kernel logic like pipes and timers. (see Figure 3.2)

Every task-structure has a bitmask, which keeps track of the currently active conditions. When a task receives a signal, the kernel updates the bitmask in the task-structure and when all defined conditions have been met, the task can be released from the condition-wait set[1]. A signal can be sent to a task at any given moment. When a task is released, the set of met conditions is cleared. This gives the kernel the ability only to start a task when there is input available and there is space available to write its output.

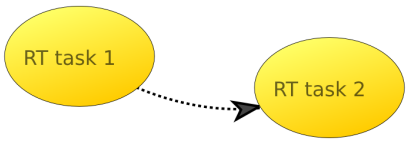

Figure 3.2: Signals: activating a task

[1] The condition-wait set will be described in Section 3.3.

Instance-independent conditions

When one task has written data into a pipe and the kernel has activated the second task by sending a signal, it does not necessarily mean that the pipe is empty after the second task is finished.

As described before, the conditions are met per task instance. To solve this problem, we introduce instance-independent conditions. When a task is released, all conditions except for the instance-independent conditions are cleared. The instance-independent conditions will stay active until someone sends a signal-clear.

3.1.3 Pipes

There are various ways to send data from one task to another. You can use shared memory and implement your own communications channel or use pipes. (see Figure 3.3)

Pipes internally have a circular buffer of which the size is specified on creation. Tasks can send data to another task (or device) by writing into a pipe. The pipe-object itself then takes care of signalling other tasks when sufficient data has been written into the pipe. In the pipe-object two signals are available: the not-full signal and the not-empty signal. The not-full signal is sent when data is read from the pipe and the amount of **free** space becomes more than *wr_threshold* bytes. The not-empty signal is sent when data is written to the pipe and the amount of **used** space becomes more than *rd_threshold* bytes. These thresholds can be configured dynamically by updating the *rd_threshold* and *wr_threshold* fields[2].

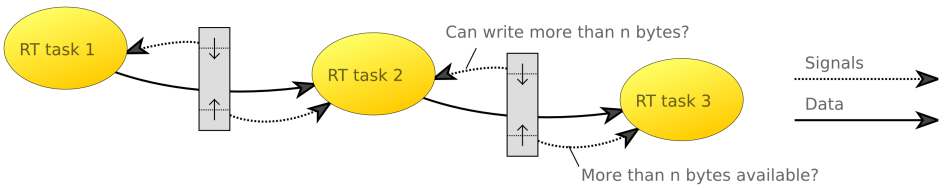

Figure 3.3: Pipes: data transfer from one task to another

To keep the kernel interface as generic as possible, all data-streams from and to the hardware have been implemented as pipes. This gives us the same flexibility as we have when using a pipe for sending data between tasks. Also, if we are switching between a software implementation of a task and a hardware implementation or Montium implementation, the task sending data to this task and the task receiving data from this task do not need another interface.

For example, Figure 3.4 shows an implementation switch. If one would dynamically switch over to the hardware implementation of task 2, one could remove task 2 from the scheduler and reconnect the pipes to and from this task to the hardware implementation.

Switching between two implementations can be done by giving the first task a pointer to the new input pipe. When all data has been read from the old pipes, these structures can be deleted. The application itself is responsible for cleaning up the old structure of pipes and tasks. The application also needs to solve the timing difference (glitch) between the old and new implementations.

[2]Changing the pipe-structure fields is discussed more in-depth in Chapter 6.

Figure 3.4: Rerouting pipes to an alternative implementation

3.2 Interrupts

When an interrupt is received by the ARM processor, the kernel starts the interrupt handler, which will handle the first part of the interrupt. Depending on what should be done by the interrupt handler, the handler can handle the complete interrupt, send a signal to one or more tasks or activate an impulse handler[3] to handle the second part of the interrupt.

3.3 Streaming-oriented Scheduling

The EDFI real-time scheduler [5] from Section 2.4.1 works with two queues and a stack. Every task is released periodically, independent of whether it has anything to do. We have based our model on this periodic task model and adapted it in such a way it can also be used for aperiodic tasks.

3.3.1 Delaying the task release

As described in Section 3.1.2, whether a task has anything to do is encoded in its conditions. While not all conditions have been met, it is a waste of time to release the task, only to find out that the task cannot do anything yet.

We can however delay the scheduling when not all conditions for the task to release have been met. Where a task should be released in the normal EDFI scheduler, it is now put on hold while not all conditions have been met. For example, the first task-bar in Figure 3.5 describes a task that is scheduled with EDFI. The second task-bar describes a task which is delayed at its third release.

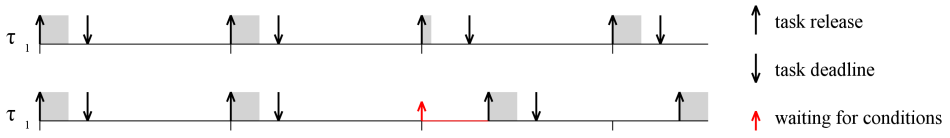

Figure 3.5: EDFI scheduling without signal-delaying and with signal-delaying

Baruah *et al.*[1] proved the following theorem:

> "If for any interval with length L, all work load offered during $[0, L]$ can be resolved before or at L, then this can be concluded for any arbitrary time interval $[t, t+L]$."

[3] See Section 2.3 for a discussion of the impulse handler.

Jansen *et al.*[5, page 3] commented on this:

> "Therefore all tasks in Ω are released simultaneously at $t = 0$, in which case they will produce the largest response time. If the tasks in Ω can make their deadlines from $t = 0$, they can make their deadlines from any point in time."

This means that tasks can be delayed without affecting the result of the EDFI schedulability analysis described in Section 2.4.2.

3.3.2 Aperiodic tasks

With this same technique, we can also implement aperiodic tasks. (Figure 3.6) An aperiodic task is scheduled as if it is a periodic task, whereby the task's period is the minimum time between two instances of the task, and the trigger to activate the task is modelled as a condition for the task to run. The worst scenario for such an aperiodic task, the scenario wherein the task is continuously activated, then resembles the scheduling of a periodic task.

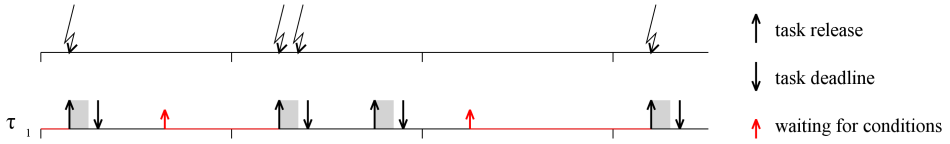

Figure 3.6: EDFI scheduling on aperiodic task

3.3.3 The scheduling model

In the scheduling algorithm, this delay is implemented by introducing a fourth group, the *condition-wait set* (see Figure 3.7). This set is inserted between the *wait queue* and the *release queue*. When a task starts a new period, the task is removed from the *wait queue* and inserted in the *condition-wait set*. When all conditions have been met, the task is released and inserted into the *release queue*.

3.3.4 The scheduler impulse handlers

The scheduler currently uses four impulse handlers. A brief description will be given for every impulse handler. The handlers are discussed from high priority to low priority.

"Task received signal"

This impulse handler checks the condition-wait set for tasks which have all their conditions met. These tasks will be removed from the condition-wait set and inserted into the released queue. The impulse handler will then activate the "Schedule" impulse handler.

"Task exit"

The currently running task called the SYS_exit() system-call. The current task is removed from the run stack and inserted into the wait queue. The impulse handler will then activate the "Schedule" impulse handler.

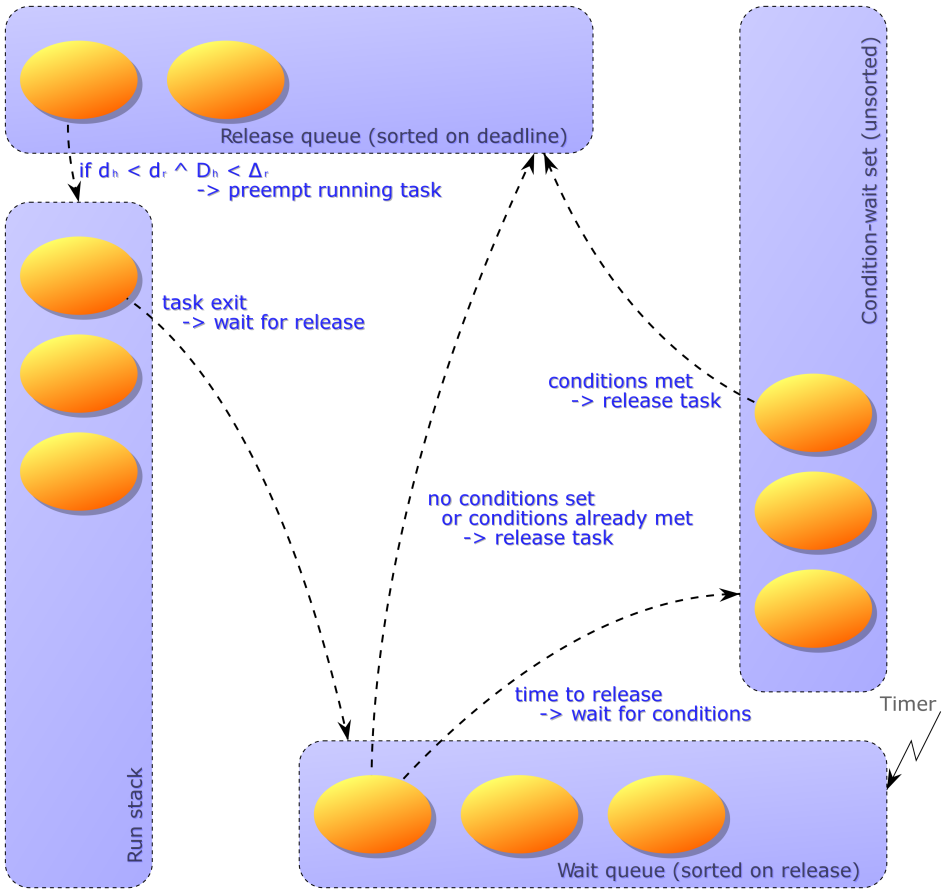

Figure 3.7: EDFI Scheduler, extended with a condition-wait set

"Received timer interrupt"

This handler is called when a task can be released from the wait queue or when a deadline has been met. The timers within the scheduler are updated. When a task in the release queue or run stack reaches its deadline, the task is moved to the wait queue. When a task from the wait queue reaches the end of its current period, the task moves to the condition-wait set or released queue. When the impulse handler is finished, it will activate the "Schedule" impulse handler.

"Schedule"

This impulse handler checks whether the currently running task (if any) should be preempted by the task on the head of the release queue.

If there are no real-time tasks available, the first task on the tdma queue is activated if it still has time left in its time-slice. When the end of the time-slice is reached, the task will be placed at the end of the tdma queue with a new time-slice.

Chapter 4

Implementation

This chapter describes the decisions and optimisations made in the implementation of the kernel.

4.1 Memory Management

As described in Section 2.2.2, the BCVP has 2 MB at **0x80000000** available for application code running on ARM1. We only can use it from approx. address **0x80008000**, because the RedBoot boot-loader uses the memory in between. If we are using the USB ELF-loader that is described in Section 6.2.3, we can use the full 2 MB.

4.1.1 Heap Memory

All memory that initially is not in use by the kernel itself is defined as heap memory. The kernel can use the heap to dynamically allocate and de-allocate memory.

A widely used heap memory algorithm is the *first free shrink algorithm* [10] [4, pg. 59-61]. On initialisation, the heap is one big free memory block, but when time passes, it becomes fragmented. The kernel keeps a linked list of free memory blocks, and when memory is needed, it searches this list for the first block that fits the size request.

A memory block header starts with a word specifying the size of the block, including the size of the header. When the memory block is free, this word is followed by a pointer to the next free memory block. The last free memory block header contains a NULL-pointer. (see Figure 4.1)

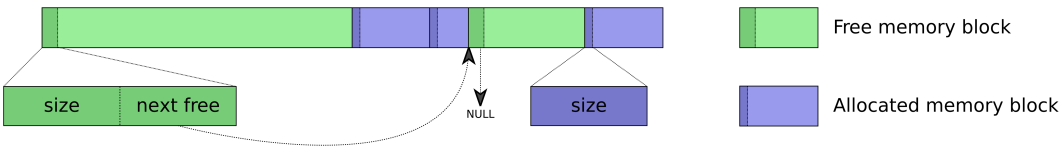

Figure 4.1: Heap memory: free memory blocks and allocated memory blocks

The algorithm used for allocating a block (Figure 4.2):

- Enlarge the requested size to a word-boundary and make sure that the required memory block can contain a free memory block header. (guarantee that there is enough space when we free this block)
- Start searching for a free memory block that is large enough to hold the requested memory.
- Split the block in two parts. The first part is the new free block and the second part is the allocated block. Only the size of the memory blocks needs to be updated, the linked list of free memory blocks doesn't change.
- Return the pointer to the data-part of the newly allocated memory block.

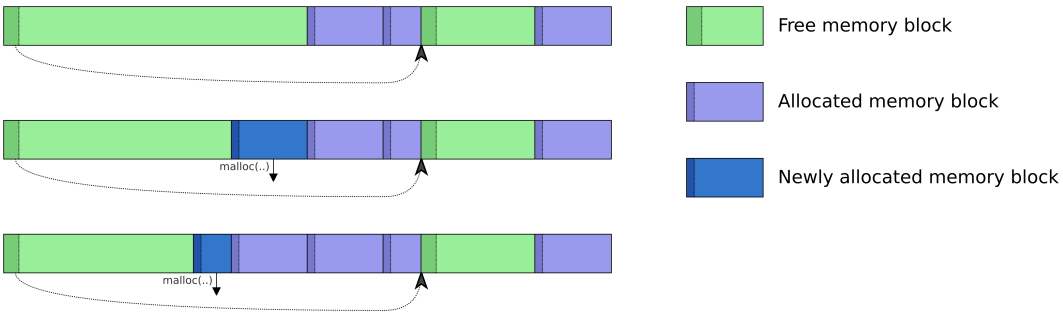

Figure 4.2: Allocating memory: two memory allocations

The algorithm used for freeing a block (Figure 4.3):

- Search the free memory blocks chain for the last free block before the given pointer. Since the first block in memory is always a free block, such a block is always available.
- Copy the next free pointer from the previous free block header to the current block header. The size the the current block header does not change when it becomes a free block.
- If the previous free block and the new free block are adjacent, merge them to prevent fragmentation of free blocks.
- If the new free block and the next free block are adjacent, also merge them.

As an optimisation on the DCOS implementation [4, pg. 59-61], the block size of a memory block in BasOS always includes the block header structures. When converting allocated memory block to a free memory block, only the linked list of free memory blocks needs to be updated.

4.2 Tasks

As mentioned before, there are two types of tasks: real-time tasks (currently scheduled with the EDFI algorithm) and non-real-time tasks (scheduled with the TDMA algorithm). Both types of tasks use the same task structure, but some fields are used differently.

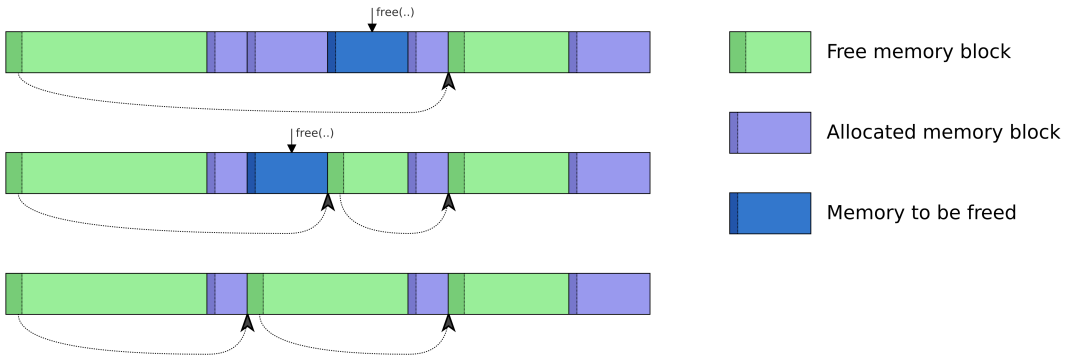

Figure 4.3: Freeing memory: two memory deallocations. on second free, two free blocks are merged

Every task in BasOS needs a task specification and a task context. The task specification (see Listing 4.1 and Table 4.1) specifies the entry-point, needed stack size, maximum runtime, the initial flags, its deadline, shared resources, etc..

For tasks with a private stack, the stack size is used for allocating enough space on the heap. For tasks with a shared stack, the value is used for guaranteeing that enough stack is available for the task to run.

Note that for non-real-time tasks, the deadline, period and resources fields are ignored. The cputime field is used for a different purpose. It specifies the length of the time- slice for the TDMA scheduler.

```
struct task_specification {
  funcptr entry;

  u32 stack_size;

  u32 deadline;
  u32 period;
  u32 cputime;

  u32 flags;

  char *name;
  task_resource_ptr *resource_sh;
  task_resource_ptr *resource_ex;
};
```

Listing 4.1: The task specification struct

The task context (see Listing 4.2 and Table 4.2) is used for storing the current state of the task. Every task that is not using the shared context object should have a private context. When entering the impulse queue runner, the current task state is stored into the task context and when leaving the impulse queue runner, a new task state is retrieved from the currently active task context.

entry	The entry-point of the task
stack_size	The max. stack size (in bytes) used by the task
deadline **period** **cputime**	The task deadline (in 1/32768 seconds) The interval at which the task is released (in 1/32768 seconds) The max. time the task will run (in 1/32768 seconds). For non-real-time tasks, this value specifies the length of the time-slice
flags	Specifies the task's type and state: - **TASK_FLAG_RT**: The task is a real-time task - **TASK_FLAG_CTXT_SHARED**: The task will share the global task context (global stack) - **TASK_FLAG_IDLING**: When set, the task is initially inserted in the condition-wait set. Otherwise, the task is initially inserted in the released queue
name	The name of the task. Used for debugging
resource_sh **resource_ex**	The shared and exclusive resources needed by this task, specified as a pointer to a NULL terminated list

Table 4.1: Description of all fields in the task specification structure

When a task is started that shares its context with already running tasks, the scheduler pushes the task state of the preempted task onto the shared stack and recycles the context object to use it for the new task. After the task has finished, the original context data is recovered from the shared stack.

```
struct task_context {
  u32 cpsr;
  u32 pc;
  u32 r[15];
#ifdef CONTEXT_MEASURE_RUNTIME
  u32 rt_runtime;
  u32 rt_lstart;
#endif // CONTEXT_MEASURE_RUNTIME
  void *heap_stack;
};
```

Listing 4.2: The task context struct

The task object itself (see Listing 4.3 and Table 4.3) contains information about its current state in the scheduler, the used condition bits (see Section 4.2.2 for more detailed information about these bits) and the tasks arguments.

cpsr	Saved state of the CPSR[a] register
pc	Saved state of the program counter (r15) register
r[15]	Saved state of the r0 ... r14 registers
rt_runtime	The total time this context is running (in 1/32768 seconds)
rt_lstart	The time when this context was last restarted
heap_stack	Pointer to the allocated memory for the stack

[a] The Current Program Status Register (CPSR)

Table 4.2: Description of all fields in the task context structure

```
struct task {
  struct task_context *context;
  u32 stack_base;
  struct task *shared_next;
  u32 flags;
  int use_count;
  int absolute;
  u32 delta;
  const struct task_specification *spec;
  u32 cond_bits;
  u32 cond_bitmask_in_use;
  u32 cond_bitmask_nonsticky;
  u32 cond_bitmask_wake;
#ifdef CONTEXT_MEASURE_RUNTIME
  u32 rt_runtime;
#endif // CONTEXT_MEASURE_RUNTIME
  void *args;
  struct task *next;
};
```

Listing 4.3: The task struct

4.2.1 Scheduling

```
static struct task * task_rt_wait_queue;
static struct task * task_rt_cond_wait_set;
static struct task * task_rt_release_queue;
static struct task * task_rt_running_stack;
```

Listing 4.4: The four EDFI-scheduler queues

As described in Section 3.3, our EDFI scheduler has three queues and one stack. They are implemented as four linked lists. (see Listing 4.4)

context	Pointer to the context object
stack_base	The stack pointer on task release
shared_next	Pointer to the preempted task with the same context object, when sharing contexts
flags	Specifies the task's type and state: - **TASK_FLAG_EXIT**: The task has ended - **TASK_FLAG_RT**: The task is a real-time task - **TASK_FLAG_DESTROY**: The task needs to be destroyed (waiting for *use_count* to reach 0) - **TASK_FLAG_CTXT_SHARED**: The task will share the global context - **TASK_FLAG_IDLING**: The task is waiting in the condition-wait set - **TASK_FLAG_WAITING**: The task is blocked. Waiting for a condition - **TASK_FLAG_ACTIVE**: The task is added to the scheduler - **TASK_FLAG_TO_BE_REMOVED**: The task should be removed from the scheduler
use_count	The number of references to this task object
absolute	The activation time relative to the previous task in the linked list or when on the head of the linked list to the *time_last_updated* variable.
delta	The inherited deadline
spec	A pointer to the task specification structure
cond_bits	The currently triggered conditions (stored as bitmask)
cond_bitmask_in_use	Bitmask of all used bits
cond_bitmask_nonsticky	Bitmask of all bits that have to be reset on task release
cond_bitmask_wake	Bitmask of all bits that wake a task from its blocking wait state
rt_runtime	The total time this task is running (in 1/32768 seconds)
args	The argument given to the entry point
next	Pointer to the next task in the queue or stack

Table 4.3: Description of all fields in the task structure

The wait queue

The *wait queue* is the queue in which all tasks wait for the moment they can be released. The queue is sorted on earliest release first. The release time can be determined by looking at the left over deadline of the task ($r_i^j + 1 = r_i^j + T_i = d_i^j + T_i - D_i$). From here, a task will move to the *condition-wait set*.

The time the task can be removed is stored as a cumulative value (over the $task \rightarrow next$ linked list) in the $task \rightarrow absolute$ field. The advantage of only storing the incremental value in the linked list is that we only have to update the first task-fields.

The condition-wait set

In the *condition-wait set*, a task will wait until all the task's defined conditions have been met. The set is implemented as a queue, which is not sorted. When all conditions have been met, the internal condition state is cleared, the deadline is activated and the task is inserted into the *release queue*. If no conditions have been defined for this task, the task will behave strict periodic and skip the *condition-wait set*.

The release queue

Just like the *wait queue*, tasks in the *release queue* are sorted, but now by their deadline. The same field $task \rightarrow absolute$ is used, but now for the cumulative deadline. On every change in the *release queue* and *run stack*, the scheduler checks if the head of the *release queue* (the task with the first deadline) can preempt the task at the head of the *run stack*. If so, the task will move from the *release queue* to the *run stack*.

If a deadline is met before the end of a task, the task immediately moves to the *wait queue*.

The run stack

The *run stack* contains the currently running real-time task on top and below it all the preempted tasks. Similar to the *wait queue* and *release queue*, the field $task \rightarrow absolute$ is stored as an incremental value. When a task ends or when its deadline is met, the task is removed from the *run stack* and inserted into the *wait queue*.

4.2.2 Signalling

Every task (see Listing 4.3) has four variables for storing the signals it is listening to and whether they have been triggered already. Due to the limited length of these field, we can only attach 32 signals to one task.

The *cond_bitmask_in_use* keeps track of which bits have been used and which haven't. The *cond_bits* contain which of these bits have been triggered, *cond_bitmask_nonsticky* tells us which bits need to be reset when a task is inserted into the release queue. The *cond_bitmask_wake* tells us on which bit we are waiting if the task is blocking (only possible in a non-real-time task).

Since the task itself doesn't keep track of the signals it is waiting for, the signal object keeps a list of tasks and its assigned bitmasks. When a signal is triggered, the kernel will loop over the task-list in the signal object and sets and resets the corresponding bits in the *cond_bits* in the task object. If all conditions have been met, which is when the *cond_bitmask_in_use* is equal to the *cond_bits*, the 'check task condition' impulse handler (which is a part of the scheduler) is triggered that will insert the task into the release queue. (see Figure 4.4)

Adding a task as a recipient for a signal is done by finding a free bit in the task's *signal_bitmask_in_use* bitmask, mark this bit as 'used' and adding the task to the list of tasks and bitmask pairs in the signal structure.

4.2.3 Feasibility Analysis

Every time a task is added to the scheduler, the scheduler checks whether the new task-set is feasible. In case of a feasible task-set, the new inherited deadlines are copied to the actual

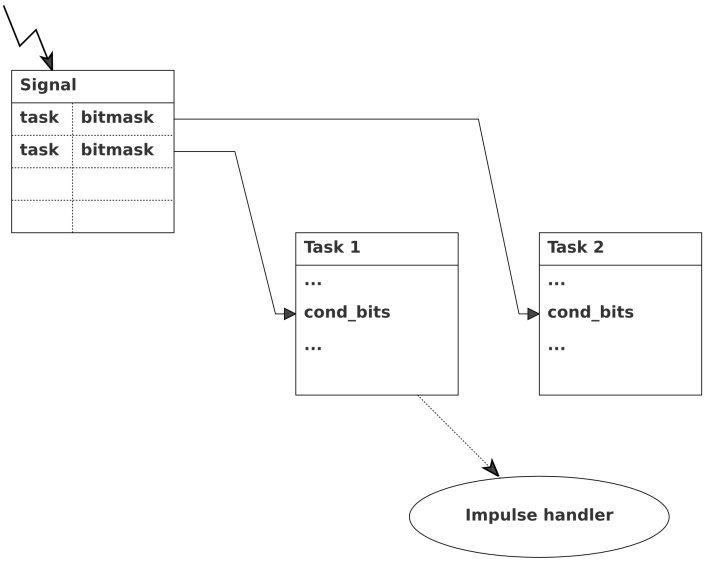

Figure 4.4: Releasing a task: Signal is triggered, task releases impulse handler

tasks and the new task is added to the scheduler.

For testing the feasibility of the new task-set, an abstract task structure is used, see Listing 4.5. The feasibility analyser replays the tasks from time $t = 0$ to $t = L$. On every task release, $W(t)$ is checked, and on every task deadline, $H(t) + C_B(t)$ is checked.

```
struct task_abs {
  const struct task_specification *spec;
  struct task_abs *next;
  struct task *task;

  int absolute;
  u32 delta;
};
```

Listing 4.5: The abstract task struct, used for testing the feasibility

Feasibility-analysis pseudo-code (actually, this code just calculates the $H(t)$, $W(t)$ and $C_B(t)$ functions as seen in Section 2.4.2):

```
feasibility_analysis(tasks):
    H=0; // workload to be resolved
    W=0; // workload offered

    queue_wait    = ();
    queue_release = tasks;

    t=0; // t_now
    loop {
        switch (task_next(queue_wait, queue_release, t, C)) {
```

```
            case TASK_RELEASE:
                // check offered load before the release
                if ((t > 0) && (W <= t)) return FEASIBLE;
                W += C;
                break;
            case TASK_DEADLINE:
                H += C;
                B = blocking_load(queue_wait, queue_release, t);
                // check to be resolved load after the deadline
                if (H + B > t) return NOT_FEASIBLE;
        }
    }
```

4.3 Interrupt Handling

The kernel can be in one of the four states *user*, *syscall*, *irq*, *impulse*. Normally, the processor is handling a task, thus the kernel resides in *user* state. When a hardware interrupt arrives at the processor or when the task uses a syscall, the state changes to respectively *irq* or *syscall*. When the interrupt or syscall is handled, but there is no need to run an impulse handler, control is returned to the same task.

When there is a need to run an impulse handler, the context of the currently running task is saved and the impulse queue runner is called. When running the impulse handlers, interrupts are in the enabled state. New hardware interrupts thus might interrupt an impulse handler, but the new hardware interrupt will not start a second impulse queue runner. When all impulses have been handled, the context of the newly running task is restored (a task switch might have occurred), and the task will continue.

All these states and transitions between the states can be found in Figure 4.5. Note that hardware interrupts can only occur when the processor is in an interrupt enabled state, which is in the *user* and *impulse* state.

4.3.1 Race-condition risks

These kernel states introduce one problem. Since the impulse handlers can be interrupted by a hardware interrupt, it is possible that both the impulse handler and the interrupt handler need to update the same kernel variables.

Normally, one would introduce semaphores or blocking mutexes in the kernel, but we wanted to keep our kernel lightweight, preferably without any possible blocking in the kernel. There are two solutions to this problem. One is to guarantee that none of the variables are accessed by both handlers simultaneously. The other option is to use atomic operations and (if necessary) disable the interrupts to update a variable.

To show what can and cannot be done, a few examples are given in Table 4.4.

When handling a system call or interrupt, only a few processor registers are saved to keep the overhead low. Only when needed, a full task state is saved. An interrupt handler can both interrupt a task and the impulse handler. In the first situation, the task state is not saved. In the second situation it is.

For allocating memory on the heap, we need to guarantee that we are the only process altering the heap control blocks. A task cannot give us these guarantees. In a system call

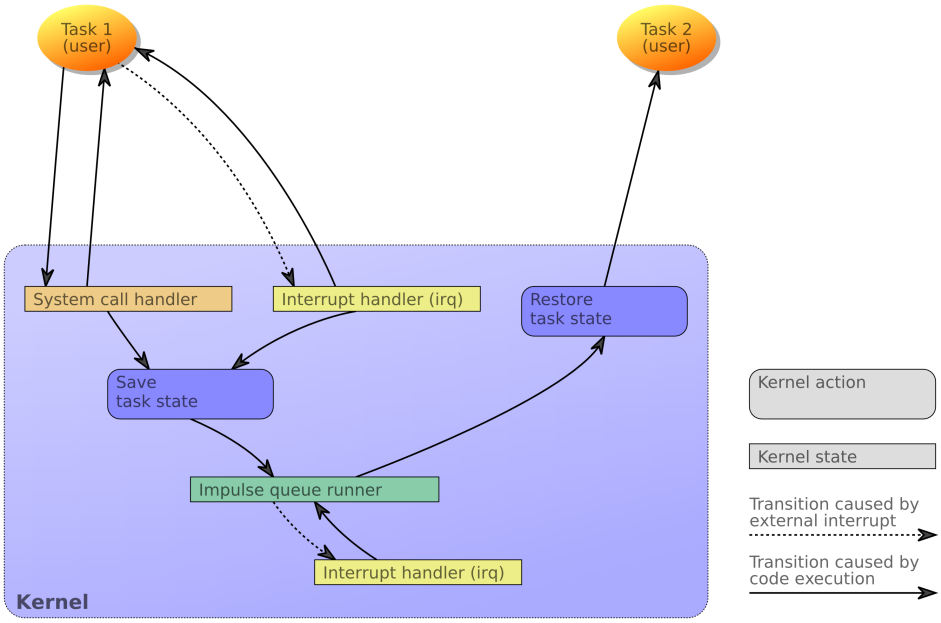

Figure 4.5: Kernel states and transitions

handler or impulse handler, we know we are the only process doing a malloc(). An interrupt handler can interrupt an impulse handler processing a malloc().

To move tasks between the task scheduling queues, we also need this guarantee. Because the current task is still active when handling a system call, we cannot change the run stack.

We must be able to update the flags field in the task structure when handling an interrupt. As a result, we have to guarantee we are updating the flags field atomically, when handling an interrupt handler. This is described in Section 4.3.1.

state	task state active	interrupts enabled	safe to use malloc()	move tasks in RT queues[a]	update task flags
user	yes	yes	via syscall	no	no
syscall	yes	no	yes	partially[c]	yes
irq	depends[b]	no	no	no	yes
impulse	no	yes	yes	yes	atomic[d]

[a] Moving tasks between the wait queue, condition-wait set, release queue and run stack
[b] Which actually just means we cannot make any assumptions about it
[c] The current task state is not saved yet. Therefore, the run stack cannot be changed.
[d] We don't like updating the *flags* field, but sometimes we just have to.
The only possibility here is to disable the interrupts temporarily.

Table 4.4: All kernel states with some characteristics

In BasOS there are two activities where we may expect race-condition problems. These are task scheduling and in-kernel debugging.

Task Scheduling

When the kernel is handling an interrupt, the *task_send_signal*() and *task_clear_signal()* functions can be called. When that happens, the *flags* and *cond_bits* will be updated and *cond_bitmask_in_use* and *cond_bitmask_wake* will be read.

Since interrupts can occur while handling impulse handlers, the impulse handler should take care that these variables are not overwritten. When we take a close look of these four fields in the task structure, we notice that only two of them, *flags* and *cond_bits*, are actually updated from the impulse handler.

The best way to update these two variables would be by using one assembler instruction which atomically sets or resets bits in memory. Unfortunately, the ARM architecture doesn't have such an instruction, so we have to disable the interrupts, update variable, and re-enable the interrupts:

```
ATOMIC(temp_variable);
task_ptr->flags |= TASK_FLAG_IDLING;
ATOMIC_END(temp_variable);
```

Note that when we are using a multi-processor environment, we probably also need to lock the bus and memory.

In-Kernel Debugging

In-kernel debugging is only used when testing or debugging a task-set or when something went wrong. We did not develop a full-blown debugger but are informed by using the printk() function. printk() writes a message to the console pipe. During the whole call, the processor is kept in an interrupt disabled state.

Chapter 5

Device Drivers

The implemented device drivers and how they are used.

5.1 USART

The BCVP has 3 USARTs. Currently, only USART0 is supported (the physical serial connection), with a bit-rate of 115200, 8 bits data, no parity and one stop bit. The task for this driver is to provide a channel which can be used for sending debugging output, without the need to run a full blown debugger.

Communication with the serial port is done via two pipes:

```
struct pipe *PIPE_usart0_in;
struct pipe *PIPE_usart0_out;
```

The implementation of the driver is very straightforward. When data is available in the *PIPE_usart0_out*, the interrupt handler that tells us we can write to the USART is unmasked. When the interrupt handler is activated, it will write one byte to the USART. When there is no more data available, the interrupt handler is masked.

The same holds for *PIPE_usart0_in*. When data can be written into the pipe, the interrupt handler that tells us we can read from the USART is unmasked. When the interrupt handler is activated, it will read one byte from the USART and writes it into the pipe. When the pipe is full, the handler is masked.

Since the USART only allows us to read or write only one character at a time, using an impulse handler would cause too much overhead.

The USART driver is initialised on boot. Messages can be sent and received immediately.

5.1.1 Code example

```
// send a message to USART0
char *msg = "Hello world!\r\n";
SYS_write(PIPE_usart0_out, msg, strlen(msg));

// receive a message from USART0
char buf[2];
SYS_read(PIPE_usart0_in, buf, 1);
```

```
// and send the character to the console
buf[1] = 0;
SYS_printf("Received character '%s'.\r\n", buf);
```

5.2 Universal Serial Bus

For faster communication with the PC, the USB driver is implemented. The driver is based on the eCos USB implementation written by one of the partners in the 4S-project, however large parts have been rewritten to better support streaming communication.

Within USB, there are 6 communication channels available. The BasOS implementation uses only 3 of them. The first channel (endpoint 0) is used for controlling the USB connection. The second channel (endpoint 1) is used for communication from the BCVP to the PC and the third channel (endpoint 2) is used for communication from the PC to the BCVP. When needed, it is possible to use endpoints 4 and 5 as extra send and receive channels.

The USB connection is configured as a full speed device (12 Mbit/sec) and for communicating via endpoint 1 and 2, bulk packets are used. The BCVP is visible as device 0x4242:0x0100, with manufacturer "4S" and product name "BCVP DiMITRI USB link". For further information about the USB protocol specification, see [18].

Both the receiving channel and the sending channel are available as pipes in BasOS:

```
struct pipe *PIPE_usb_in;
struct pipe *PIPE_usb_out;
```

BasOS uses the same signalling technique as described for the USART0, except that it now reads and writes blocks of 64 bytes (the max. amount of bytes in a frame). We use the peripheral double buffering if the last frame was exactly 64 bytes. If the last frame had fewer bytes, using the double buffer may cause the controller to get confused.

The USB driver is initialised on boot. Messages can be sent and received when the USB device is configured. The kernel will notify the the task with the *SIG_usb_configured* signal.

On the PC side, libusb [16] can be used for sending to and receiving from the BCVP device.

5.2.1 Code example

```
// wait until the USB device is configured
SYS_signal_wait(&SIG_usb_configured);

// send a message to USB
char *msg = "Hello world!\r\n";
SYS_write(PIPE_usb_out, msg, strlen(msg));

// receive a message from USB
char buf[2];
SYS_read(PIPE_usb_in, buf, 1);

// and send the character to the console
```

```
buf[1] = 0;
SYS_printf("Received character '%s'.\r\n", buf);
```

5.3 The routing network and Montium processors

The implementation of the driver for configuring the routing network and Montium processors started with the blocking implementation from Erik van der Sluis (see [9]). The code is rewritten into a stateful non-blocking form, which is suitable for being used in interrupt handling. There are four input pipes and four output pipes available:

```
struct pipe *PIPE_montium_in[4];
struct pipe *PIPE_montium_out[4];
```

These pipes initially are not connected in the routing network. Each pipe first needs to be connected with several calls to *SYS_router_configure()*. See Figure 5.1 for the routing network to the montium processors. Currently, the montiums and their network are emulated on an FPGA. The Xilinx Virtex-II 3000 can hold one router and one montium, while the Virtex-II 8000 can hold two routers and three montiums.

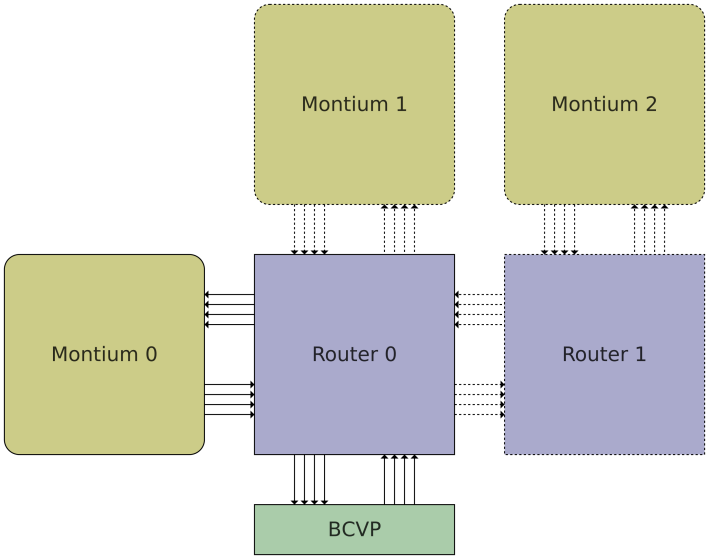

Figure 5.1: The Montium routing network

5.3.1 Configuration of the montium lanes

The montium pipes internally use 16 bits values, combined with a 2 bit *flit*, which specifies the packet type. This is not the most convenient message format to be used for sending data. To this aim, we need to configure the pipes first for their data formats.

One can choose to send the data with or without a flit type. When sending without a flit type, the type should be specified in the config. It is also possible to combine two lanes.

The pipes can be configured with the *SYS_conf_lane_in*(*lane*, *config*) and *SYS_conf_lane_out*(*lane*, *config*) system calls.

The lane number matches the index in the PIPE_montium_in[] and PIPE_montium_out[]. On boot, all lanes are configured as disabled (LANE_CONF_DISABLED).
There are four data-formats available:

LANE_CONF_16_BIT
: The pipe uses a single lane, which has a width of 16 bits. data is sent with one predefined flit type.

LANE_CONF_16_AND_FLIT
: The pipe uses a single lane, which has a width of 16 bits. The data within the pipe is 32 bits wide. The data is encoded in bits 0..15. Bits 16 and 17 specify the flit type.

LANE_CONF_32_BIT
: The pipe uses two lanes, which combined have a width of 32 bits. data is sent with one predefined flit type.

LANE_CONF_32_AND_FLIT
: The pipe uses two lanes, which combined have a width of 32 bits. The data within the pipe is 64 bits wide. The data is encoded in bits 0..31. Bits 32 and 33 specify the flit type.

When we need to predefine a flit type, the type is added to the *config* value. The four flit types are:

- LANE_CONF_FLIT_DATA
- LANE_CONF_FLIT_ADDR
- LANE_CONF_FLIT_TAIL
- LANE_CONF_FLIT_CMD

There are only four lane combinations possible. These are:

- Lane 0 with lane 1 (specified with LANE_CONF_DUAL_LANE_H)
- Lane 2 with lane 3 (specified with LANE_CONF_DUAL_LANE_H)
- Lane 0 with lane 2 (specified with LANE_CONF_DUAL_LANE_V)
- Lane 1 with lane 3 (specified with LANE_CONF_DUAL_LANE_V)

The driver uses only the first pipe when reading or writing. The second pipe is ignored while being configured in dual mode.

For configuring the montium processors itself, see [2].

5.3.2 Code example

```
// configure the routing network
// - connect montium 0 lane 0 to bcvp lane 0
SYS_router_config(0, 1, 0,0, 2,0);
// - connect montium 0 lane 1 to bcvp lane 1
SYS_router_config(0, 1, 0,1, 2,1);
```

```
// - connect bcvp lane 0 to montium 0 lane 0
SYS_router_config(0, 1, 2,0, 0,0);
// - connect bcvp lane 1 to montium 0 lane 1
SYS_router_config(0, 1, 2,1, 0,1);

// prepare lane 0 for uploading the montium configuration data
SYS_conf_lane_out(0, LANE_CONF_16_AND_FLIT);

// send configuration data
SYS_write(PIPE_montium_out[0], montium_code, sizeof(montium_code));

// send 'run' command
unsigned int cmd = 0x30000 | HYDRA_CMD_RUN;
SYS_write(PIPE_montium_out[0], &cmd, 4);

// prepare lane 0+1 for sending and receiving data
// - configure input lanes
SYS_conf_lane_in(0, LANE_CONF_32_BIT | LANE_CONF_FLIT_DATA | LANE_CONF_DUAL_LANE_H);
// - configure output lanes
SYS_conf_lane_out(0, LANE_CONF_32_BIT | LANE_CONF_FLIT_DATA | LANE_CONF_DUAL_LANE_H);

// send a message via lane 0+1 to Montium 0
unsigned int msg = 0x12345678;
SYS_write(PIPE_montium_out[0], &msg, 4);

// receive data from Montium 0
unsigned int buf;
SYS_read(PIPE_montium_in[0], &buf, 4);

// and send the character to the console
SYS_printf("Received data-word 0x%x.\r\n", buf);
```

Chapter 6

Examples

This chapter gives an overview of how to write applications for BasOS.

6.1 Writing Applications

6.1.1 Tasks, Signals, Pipes, Resources

A task

This chapter focuses on how to write a simple application in BasOS. The first task, a non-real-time task, is started by the kernel. The task's name is *__start__*. A task is represented in a C function and when this function returns, **SYS_exit**() is automatically called.

In Listing 6.1, a simple message is sent to the standard output channel (normally the serial port), and then exits.

```
void __start__( void )
{
  /* task implementation */
  SYS_strprint("Hello world!\n");
}
```

Listing 6.1: A simple implementation of a task

Creating our own task

To create a task, we need a filled-in task specification struct. For real-time tasks, a deadline, period and cputime should be given, so the kernel can check whether the new task-set is feasible. For non-real-time tasks only the time-slice length is needed, which is filled in in the *cputime* field. (see Listing 6.2)

The creation and activation of the task is split in two system calls, so we can attach the task to certain signals before activating the task.

```
void task_1( void )
{
  /* task implementation */
```

```
  SYS_strprint("Another second has passed..\n");
}

static const struct task_specification task_1_spec = {
  .entry          = &task_1,
  .stack_size     = 256,    // the needed stack size in bytes
  .deadline       = 0x2000, // 250 ms
  .period         = 0x8000, // 1 second
  .cputime        = 0x1000, // 125 ms
  .name           = "task 1",
  .flags          = TASK_FLAG_RT | TASK_FLAG_IDLING,
                            // mark the task as a real-time task,
                            // starting in the condition-wait state.
  /* no shared resources */
  .resource_sh    = NULL,
  .resource_ex    = NULL,
};

void
__start__( void )
{
  struct task *task_1 = SYS_task_create( &task_1_spec );
  SYS_scheduler_add_task( task_1 );
}
```

Listing 6.2: Creating a task

Adding a shared resource

A list of shared resources can be given. The scheduler guarantees that a task with an exclusive lock to a specific resource will not be preempted by another task that uses the same resource.

In this example (see Listing 6.3), both tasks are activated at the same moment and since both tasks have the same deadline, it is possible that on one cycle task_1 is called first and on the other task_2.

```
int current_time = 0;
struct task_resource shared_resource_1 = { .ptr = &current_time };

void task_1( void )
{
  current_time++;
}

void task_2( void )
{
  SYS_printf("%d seconds have elapsed..\n", current_time);
}
```

```
static const struct task_specification task_1_spec = {
  .entry          = &task_1,
  .stack_size     = 256,
  .deadline       = 0x2000, // 250 ms
  .period         = 0x8000, // 1 second
  .cputime        = 0x1000, // 125 ms
  .name           = "task 1",
  .flags          = TASK_FLAG_RT | TASK_FLAG_IDLING,
  .resource_sh    = NULL,
  .resource_ex    = (task_resource_ptr[]){ &shared_resource_1, NULL },
    // list of exclusive resources. When the task is running, only this
    // task is allowed to use these resources.
    // the list is terminated with a NULL value.
};

static const struct task_specification task_2_spec = {
  .entry          = &task_2,
  .stack_size     = 256,
  .deadline       = 0x2000, // 250 ms
  .period         = 0x8000, // 1 second
  .cputime        = 0x1000, // 125 ms
  .name           = "task 2",
  .flags          = TASK_FLAG_RT | TASK_FLAG_IDLING,
  .resource_sh    = (task_resource_ptr[]){ &shared_resource_1, NULL },
    // list of shared resources. When the task is running, the resource
    // can be shared with other real-time tasks.
    // the list is terminated with a NULL value.
  .resource_ex    = NULL,
};

void
__start__( void )
{
  struct task *task_1 = SYS_task_create( &task_1_spec );
  struct task *task_2 = SYS_task_create( &task_2_spec );

  SYS_scheduler_add_task( task_1 );
  SYS_scheduler_add_task( task_2 );
}
```

Listing 6.3: Two tasks with a shared resource

Signals

When we define a condition for task_2, this task will stay in the condition-wait scheduling set until it receives signal *sig* from task_1 (see Listing 6.4). Tasks can be dynamically attached

to a signal with the *SYS_signal_add_listener*() call.

```
struct signal *sig;

void task_1( void )
{
  /* task 1 implementation */
  ...

  /* send signal to task_2 */
  SYS_signal_set( sig );
}

void
__start__( void )
{
  /* create tasks */
  ...

  /* create signal */
  sig = SYS_signal_create();
  SYS_signal_add_listener( sig, task_2 );

  /* add tasks to scheduler */
  ...
}
```

Listing 6.4: Sending a signal from task_1 to task_2

Pipes

Instead of using shared memory, one can use a pipe (see Listing 6.5 and Appendix D). When there are four or more bytes available in the pipe, task_1 is signalled, and when there are four or more bytes to be read, task_2 is signalled. The signals stay enabled until the pipe is too full or too empty.

```
struct pipe *pipe;

void task_1( void )
{
  static int current_time = 0;
  current_time++;
  SYS_write_nb(pipe, &current_time, sizeof(int));
}

void task_2( void )
{
```

```
  int time;
  SYS_read_nb(pipe, &time, sizeof(int));
  SYS_printf("%d seconds have elapsed..\n", time);
}

void
__start__( void )
{
  /* create tasks */
  ...

  /* create pipe */
  pipe = SYS_pipe_create(7); // create a 7 byte pipe
  pipe->rd_threshold = 4;
  pipe->wr_threshold = 4;
  SYS_signal_add_listener( &(pipe->sig_wr), task_1 );
  SYS_signal_add_listener( &(pipe->sig_rd), task_2 );

  /* add tasks to scheduler */
  ...
}
```

Listing 6.5: Using pipes instead of shared memory

6.1.2 System Calls

System calls are called with the ARM *swi* instruction. Due to the ARM architecture, this instruction provides us a fast and reliable method for system call handling. The system call handler is run in the interrupt disabled state. When the system call triggers an impulse handler (e.g. *SYS_exit*()), a task switch can occur. (see also Figure 4.5)

There are 26 system calls available. They are all described in Appendix C.

6.2 Tools

For better configuration and utilisation of the kernel, several tools have been written. These tools are described in this section.

6.2.1 Stack usage prediction

When we would like to make optimal use of the kernel, we would probably put the stack in the DTCM (see Section 2.2.2). Since there is not much space, we would like to minimise the needed stack size.

By following the flow of an application and counting all stack operations (also described in [8]), we can determine in most cases the needed stack size. Because a snippet of C code is not always compiled the same way, we are not analysing the C code, but the generated assembly code.

The simulation starts at the task's entry point. Every instruction is then simulated, and the lowest value of the stack register (r13) is stored (the stack grows downward). When a conditional instruction is found, both code paths are simulated. After all possible code paths have been processed, the lowest value of the stack register is known. The initial stack value minus the lowest stack value gives us the maximum needed stack.

Unfortunately, using this simulation method doesn't work in all situations. Sometimes the simulation will take infinite time in case of recursive functions. Also, the simulator may not know where the application continues. For example when using function pointers or optimised switch statements.
A list of known problems:

- *alloca(size)*
  The alloca function allocates *size* bytes of space in the stack frame of the caller. If *size* is a constant, we can easily determine the needed amount of stack. But often, the *size* depends on other variables, which makes determination of the stack size impossible without the use of backtracking techniques.

- **indirect calls** (e.g. using variables for function pointers)
  We normally do not keep track of read-writable memory. Therefore we don't know which pointer was written to a specific memory location. Again, we probably need back-tracking techniques to determine where the program continues.

- **recursive functions**
  Since we follow all possible task flows, a recursive function will be call infinitely. This would result in an infinite stack size. The only way to solve this problem is to know which conditional expression should be watched to know the max. amount of recursions.

- **jump-tables** (often created by compiler optimisations)
  Jump-tables use a register as index into the table. We know the offset of the jump-table, but we don't know the length. Previous instructions have to be analysed further to know which values the register can have.

6.2.2 Dynamic Application Loading

When loading multiple applications into the system and we are not using virtual memory, all tasks (the code itself) should be loaded on disjoint memory addresses. Even when the memory address is specified at compile-time, it is possible that the given memory address range is no longer free.

Therefore we would like to be able to load an application at any given location at runtime, so we can just malloc() some memory, load the image into this memory and start the task.

When looking at an ELF object file [13], we actually find all information which is necessary to build such an image. In an ELF file, every .text section contains one or more functions and each section has a list of relocation records.
A relocation record contains the following information:

- The type of relocation record.

- The offset in the current section.

- The symbolic name.

A dynamic image in BasOS is normally started with the *__start__* function. If we look at the relocation records for this function, we notice other sections we have to include. These sections include yet another sections, etc.. Finally, a list of all needed sections is known and an image can be built by concatenating these specific sections.

Now, only the relocation records need to be fixed. All the symbols are known, so we know their offset in the produced image. For ARM binaries, usually only the **R_ARM_ABS32** and **R_ARM_PC24** are used:

- **R_ARM_ABS32** relocations (absolute address, 32 bit)
  These relocations point to a given symbol. The symbol points to a section and an offset. The offset relative to the start of the new dynamic image is then the offset of the section in the new image plus the offset of the symbol itself.

  We can update the new image itself at the offset the relocation record is pointing to. Then only the base address of the image itself should be added at loading.

- **R_ARM_PC24** relocations (program counter, 24 bit)
  Program counter relocations are relative to their callers offset and not relative to the absolute address the image is loaded on. We can just calculate the difference in offset and add this to the offset the relocation points to. After updating the new image, no further processing has to be done.

Image Format

We now have an image containing all needed sections and we have a list of all offsets within the image to which the base offset has to be added. With this information we can produce a dynamic image object, which can easily be loaded on any given offset in memory.

offset	**type**	**description**
0	word	signature (always 1889875327)
4	word	version (always 0x00010002)
8	word	total size in memory
12	word	total size in image (n)
16	byte[n]	image data
$16 + n$	word	reloc size (m)
$20 + n$	word[m]	reloc pointers
$24 + n + m$	word	entry point
$28 + n + m$	word	stack size

6.2.3 Second Stage USB Boot-loader

Loading an image by using RedBoot had several problems:

- Uploading with a max. speed of 115200 bits/sec is really slow.
- We cannot use the first 30 KB of memory for our image. The RedBoot loader uses this space.
- The RedBoot loader doesn't seem to check where we are writing and if we are allowed to write there.

- Most compilers assume that the .bcc section is zeroed by the ELF-loader. The RedBoot loader doesn't do this.
- Occasionally, the uploading via RedBoot stops and we have to restart uploading the image.

Therefore a new boot-loader was written which doesn't have these problems. By using the tightly-coupled memory (32 KB code, 64 KB data), the complete 2 MB (ARM1 memory) and 1 MB (ARM0 memory) can be used for loading an ELF image. [13]

The loader uses the USB driver described in Section 5.2. After loading the USB boot-loader, the PC first sends the file size, then sends the ELF file. The loader parses this, loads the image in memory and starts the execution.

Chapter 7

Recommendations

In this chapter we would like to emphasise the parts of the kernel that are not yet fully developed. Some parts have a partly working implementation, other do not. We have listed the recommendations for a future kernel implementation, as well as some recommendations for next revisions of the currently used hardware.

7.1 Memory management

7.1.1 Memory Allocation Algorithm

Most kernel structures are allocated on the heap.

When using the first free fit and shrink heap memory algorithm, the allocate function will probably exhaust the first free block when random block allocations are done. This may result in high memory fragmentation. The longer the kernel runs, the slower the memory allocation function will be. There is no maximum time defined of how long a memory allocation may take.

The chosen algorithm is simple, we did not spend much time on improvements.

7.1.2 Splitting memory in kernel-memory and application-memory

The BCVP both has tightly-coupled memory and normal memory. The latter one is of-course slower than the first. Since kernel overhead is inevitable (e.g. hardware interrupt handling), it might be interesting to minimise this overhead by using the faster tightly-coupled memory for kernel tasks.

7.2 Scheduler

7.2.1 Other scheduling algorithms

The currently used scheduler can both schedule real-time tasks as non-real-time tasks. Real-time tasks are scheduled with a variant on the EDFI algorithm [5] and in its slack time non-real-time with TDMA (Time-Division Multiple Access).

Rate Monotonic, Deadline Monotonic

Other real-time schedulers of interest are the static priority Rate Monotonic and Deadline Monotonic schedulers. They are a good alternative to the Earliest Deadline First algorithm.

Sporadic Server, Total Bandwidth Server

Another thing is that we are combining the EDFI algorithm with the TDMA algorithm. The currently implemented non-real-time tasks are to be considered as asynchronous tasks without any real-time requirements. In general, they are used for additional system support such as initialisation.

There are other scheduling algorithms that can deal with these and other asynchronous tasks, which introduces new interesting scheduling possibilities which are not deployed so far in BasOS. (e.g. Sporadic Server, Total Bandwidth Server)

7.3 Drivers

7.3.1 Usage of PDCs

Some peripherals have implemented a Peripheral Data Controller (PDC). This helps the processor with sending data from memory to this device and vice versa. The pipe mechanism can be extended to have support for these PDCs.

Of the currently implemented drivers, the USART0 has a PDC. It is so far unclear whether the USB device is also equipped with a PDC.

7.3.2 Montium

16 bit circular buffer access

The current memory-mapped interface gives us a circular buffer with 16-bits values, but it is addressed at 32-bits offsets. As a result, the ARM cannot use optimised block-reads and writes. A 16-bit wide circular buffer data would improve this.

Peripheral Data Controller

As described in Section 7.3.1, some peripherals have a PDC, which helps the kernel in moving data between memory and a peripheral. The routing lanes to the Montiums do not have such an interface. Adding a PDC controller to the Montium lanes would improve the data transfer to and from the Montiums.

7.3.3 Implement more drivers

The BCVP has a long list of devices. Currently, only a few drivers have been implemented. Other peripherals lack driver support. Here follows a list of peripherals without drivers.

- Real Time Clock
- Triple DES
- Serial Synchronous Controller

- Serial Peripheral Controller
- Digital Down Converter
- Two-wire Interface
- Ethernet MAC
- Multimedia Card Interface
- Viterbi Decoder
- LCD Controller

7.4 Usage of Cyclic Asynchronous Buffers

Although this is probably more something for a task itself to arrange, it would still be nice to have a standardised library for accessing Cyclic Asynchronous Buffers (CABs). With the combination of CABs and shared memory, tasks can probably communicate much faster than using the currently deployed pipe system calls.

Chapter 8

Conclusions

A real-time EDFI kernel has been developed and the design goals are met. reached. The kernel uses a minimal amount of memory. The kernel fits in the Tight Coupled Memory and has dynamic memory management. For interrupts, impulse handling is used, which results in efficient interrupt handling with considerable improvement of interrupt disable times.

The kernel uses EDFI scheduling for real-time tasks, with an optimisation for streaming applications. Non-real-time tasks are scheduled with the TDMA scheduling algorithm. The kernel is able to dynamically add and remove tasks, which makes dynamic application loading possible. For this dynamic task loading, a tool has been written, which makes it possible to load the application at any given memory offset.

For fast kernel loading, an USB boot-loader has been written, which makes it possible to upload new kernel images with a speed of 1 MB/sec.

For interaction with the kernel, a user-friendly API is chosen. We use pipes and signals for communication between tasks and peripherals. We expect the kernel to give good support for streaming applications. In particular we currently provide a well appreciated support for the Montium processor.

Bibliography

[1] S.K. Baruah, A.K. Mok and L. Rosier, '*Preemptively scheduling hard-real-time sporadic tasks on one processor*', Proceedings of the Real-Time Systems Symposium, Dec 1990, pp. 182-190

[2] M.D. van de Burgwal, '*Hydra Protocol Specification*', Design document for implementing of the Hydra CCU, August 21, 2005, University of Twente. *(not published)*

[3] Giorgio C. Buttazzo, '*Hard Real-Time Computing Systems*', Predictable Scheduling Algorithms and Applications, 3rd printing, Kluwer Academic Publishers, ISBN `0-7923-9994-3`

[4] Tjerk J. Hofmeijer, '*The development of system software to support a data centric real-time architecture for sensor networks*', Master Thesis, July 2004, University of Twente.

[5] P.G. Jansen, S.J. Mullender, P.J.M. Havinga, H. Scholten, '*Lightweight EDF Scheduling with Deadline Inheritance*', `http://doc.utwente.nl/fid/1145`.

[6] P.M. Heysters, G.J.M. Smit, E. Molenkamp, '*A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems*', The Journal of Supercomputing, Volume 26, Number 3, November 2003, pages 283-308, ISSN `0920-8542`

[7] Maurice L.M. Luttmer, Han Ribbers, Pierre G. Jansen, '*Getting short interrupt disable times by Impulses*', Memoranda Informatica 89-28, April 1989, University of Twente, ISSN `0923-1714`

[8] John Regehr, '*Say no to stack overflow*', Embedded Systems Design, `http://www.embedded.com/showArticle.jhtml?articleID=47101892`.

[9] Erik van der Sluis, '*Communication and control on a tiled architecture*', November 23, 2006, University of Twente.

[10] Nick Barnes, Richard Brooksby, David Jones, Tony Mann, Gavin Matthews, *et al.*, '*The Memory Management Reference* ', `http://www.memorymanagement.org/articles/alloc.html` .

[11] Smart Chips for Smart Surroundings, `http://www.smart-chips.org/`.

[12] ARM Documentation, `http://www.arm.com/documentation/`.

[13] '*ARM ELF*', SWS ESPC 0003 A-08, 22 september 1999, Engineering Software Group, Development Systems Business Unit, http://netwinder.osuosl.org/pub/netwinder/docs/arm/ARMELFA08.pdf.

[14] eCos embedded operating system, http://ecos.sourceware.org/.

[15] eCos RedBoot bootstrapping environment, http://ecos.sourceware.org/redboot/.

[16] libusb Documentation, http://libusb.sourceforge.net/.

[17] TinyOS – An open-source OS for sensor networks, http://www.tinyos.net/.

[18] USB 2.0 Reference, http://www.usb.org/developers/docs/.

Appendix A

Acronyms

AHB	Advanced High-Speed Bus
AIC	Advanced Interrupt Controller
API	Application Programming Interface
APMC	Advanced Power Management Controller
BCVP	Basic Concept Verification Platform
CAB	Cyclic Asynchronous Buffers
CCU	Communication and Configuration Unit
CPSR	Current Program Status Register
CPU	Central Processing Unit
DBGU	Debug Unit
EDF	Earliest Deadline First
EDFI	Earliest Deadline First with deadline Inheritance
ELF	Executable and Linkable Format
FIQ	Fast Interrupt Request
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
HiCVP	Highly integrated Concept Verification Platform
IRQ	Interrupt Request
MC	Memory Controller
PDC	Peripheral Data Controller
PIO	Programmable IO Controller

RAM	Random Access Memory
ROM	Read Only Memory
RTC	Real-Time Clock
RTOS	Real-Time Operating System
SPSR	Saved Program Status Register
ST	System Timer
SWI	Software Interrupt
TC	Timer Counter
TCM	Tightly-coupled Memory
TDMA	Time Division Multiple Access
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Appendix B

GCC Cross-compiler build script

```
#!/bin/bash

# halt on error
set -e

#download location
# lynx 'http://www.gnuarm.com/'

mkdir tmp
cd tmp

PREFIX="$HOME/gnu-arm/toolchain"
mkdir -p "$PREFIX"

CONF_ARGS="--target=arm-elf --enable-interwork --enable-multilib"
CONF_ARGS="$CONF_ARGS --with-cpu=arm9 --without-fp"
GCC_VER=3.4.3
BINUTILS_VER=2.15
NEWLIB_VER=1.12.0

export PATH="$PREFIX/bin:$PATH"

# Build binutils
tar jxf ../binutils-$BINUTILS_VER.tar.bz2
(
    # patch readelf binary:
    #   readelf will display full section names when verbose flag is set.
    #   Newer versions of readelf can show full section names with the
    #   --section-details parameter.
    cd binutils-$BINUTILS_VER
    patch -p1 < ../../binutils_readelf_hack.diff
)

mkdir binutils-build
cd binutils-build
../binutils-$BINUTILS_VER/configure --prefix="$PREFIX" $CONF_ARGS
make all
```

```
make install
cd ..

# Build gcc (with newlib support)
tar jxf ../gcc-$GCC_VER.tar.bz2
tar zxf ../newlib-$NEWLIB_VER.tar.gz
mkdir gcc-build
cd gcc-build
../gcc-$GCC_VER/configure --prefix="$PREFIX" $CONF_ARGS \
    --enable-languages="c,c++" \
    --with-newlib --with-headers=../newlib-$NEWLIB_VER/newlib/libc/include
make all-gcc
make install-gcc
cd ..

# Build newlib
mkdir newlib-build
cd newlib-build
../newlib-$NEWLIB_VER/configure --prefix="$PREFIX" $CONF_ARGS
make all
make install
cd ..

cd gcc-build
make all
make install
cd ..
```

Appendix C

Kernel API

task control

SYS_task_create

Synopsis:

struct task * **SYS_task_create**(*const struct task_specification* * spec)

Description:

Dynamically allocates and initialises a task structure. In case the task has its own stack, a context structure will also be initialised and attached to the task structure.

Return value:

On success, a new task structure is returned. The task is completely initialised and can be added to the scheduler directly.

On error, NULL is returned. In this case, there was not enough heap memory available.

SYS_task_destroy

Synopsis:

void **SYS_task_destroy**(*struct task* * task)

Description:

Marks the task as 'to be destroyed'. When there are no more references to the task structure, the task is freed (implemented with a use-counter). If a context structure is allocated for this task, the context structure is also freed.

SYS_scheduler_add_task

Synopsis:

int **SYS_scheduler_add_task**(*struct task* * task)

Description:

Adds the given task structure to the currently active task-set. If the task is a real-time task, a feasibility analysis is done. If the task is flagged as not being idle, the task will be activated.

Return value:

On success, 0 is returned. On error, -1 is returned. In this case, the feasibility analysis failed.

SYS_scheduler_remove_task

Synopsis:

void **SYS_scheduler_remove_task**(*struct task* * task)

Description:

Marks the given task as 'to be removed from the scheduler'. The scheduler will remove the task from the scheduler when it stumbles upon the task.

SYS_exit

Synopsis:

void **SYS_exit**()

Description:

Stops the currently running task. If the currently running task is a periodic real-time task, the task will move to the wait queue. Otherwise, the task will automatically be destroyed on exit.

If the main task-function returns, **SYS_exit** is automatically called.

memory management

SYS_malloc

Synopsis:

void * **SYS_malloc**(*int* size)

Description:

Allocates *size* bytes of memory and returns a pointer to the allocated memory. The memory is not cleared.

Return value:

On success, a pointer to the allocated memory is returned.

On error, NULL is returned. In this case, the requested memory size was 0 bytes or there was not enough heap memory available.

SYS_realloc

Synopsis:

void * **SYS_realloc**(*void* * ptr, *int* size)

Description:

Attempts to resize the *ptr* to *size* bytes. If that fails, a new memory block is allocated, all data is moved to the new memory, and a pointer to the allocated memory is returned.

Calling **SYS_realloc** with NULL as *ptr* is equal to calling **SYS_malloc**(size). Calling **SYS_realloc** with 0 as *size* is equal to calling **SYS_free**(ptr).

RETURN VALUE:

On success, a pointer to the allocated memory is returned. If the address has changed, the old memory location is freed.

On error, NULL is returned. In this case, there was not enough heap memory available. (if the newly requested size is 0, it is actually not an error. The old memory location is freed)

SYS_free

SYNOPSIS:

void **SYS_free**(*void* * ptr)

DESCRIPTION:

Frees the previously with **SYS_malloc** or **SYS_realloc** allocated memory. When given a NULL-pointer, nothing is done.

signal handling

SYS_signal_create

SYNOPSIS:

struct signal * **SYS_signal_create**()

DESCRIPTION:

Dynamically allocates and initialises a signal structure.

RETURN VALUE:

On success, a new signal structure is returned.

On error, NULL is returned. In this case, there was not enough heap memory available.

SYS_signal_destroy

SYNOPSIS:

void **SYS_signal_destroy**(*struct signal* * signal)

DESCRIPTION:

Unregisters all registered tasks in this signal and frees the signal structure.

SYS_signal_add_listener

SYNOPSIS:

void **SYS_signal_add_listener**(*struct signal* * signal, *struct task* * task)

DESCRIPTION:

Adds a signal to a task. When all conditions, that are set by receiving such a signal, on this task have been met, the task is released.

SYS_signal_remove_listener

Synopsis:

void **SYS_signal_remove_listener**(*struct signal* * signal, *struct task* * task)

Description:

Unregisters the task in the signal structure.

SYS_signal_set

Synopsis:

void **SYS_signal_set**(*struct signal* * signal)

Description:

Trigger a signal. All tasks in the signal structure are notified, and when one of these tasks was in the condition-wait state, whereby this signal is the last signal to trigger the task, the task is released.

SYS_signal_reset

Synopsis:

void **SYS_signal_reset**(*struct signal* * signal)

Description:

"Untrigger" a signal. Only works on stateful signals.

SYS_signal_wait

Synopsis:

void **SYS_signal_wait**(*struct signal* * signal)

Description:

Wait until a signal is triggered. This should only be used in non-real-time tasks.

pipes

SYS_pipe_create

Synopsis:

struct pipe * **SYS_pipe_create**(*int* size)

Description:

Dynamically allocates an initialises a pipe with a buffer of *size* bytes.

Return value:

On success, a pointer to the pipe structure is returned.

On error, NULL is returned. In this case, there was not enough heap memory available.

SYS_pipe_destroy

Synopsis:

void **SYS_pipe_destroy**(*struct pipe* * pipe)

Description:

The signals defined in the pipe will be unregistered and freed. The buffer in the pipe structure is freed, and finally the pipe structure itself is freed.

SYS_read_nb

Synopsis:

int **SYS_read_nb**(*struct pipe* * pipe, *char* * buffer, *int* buffer_length)

Description:

Reads from a pipe. The system call will not block and return immediately. When there are fewer bytes available than *buffer_length*, the available bytes are read.

When there is enough space available that another task can write into this pipe, the 'not-full' signal is triggered.

Return value:

Returns the number of bytes read from the pipe.

SYS_write_nb

Synopsis:

int **SYS_write_nb**(*struct pipe* * pipe, *const char* * buffer, *int* buffer_length)

Description:

Writes to a pipe. The system call will not block and return immediately. When less bytes than *buffer_length* can be written to the pipe, only these bytes are written to the pipe.

When there is enough data available that another task can read from this pipe, the 'not-empty' signal is triggered.

Return value:

Returns the number of bytes written to the pipe.

montium specific system-calls

SYS_router_configure

Synopsis:

void **SYS_router_configure**(*short* router, *short* status, *short* port_in, *short* lane_in, *short* port_out, *short* lane_out)

Description:

With this command a connection within a router can be made or broken. To communicate with a montium processor, a path through the router should first be planned before one can reach the montium. For more information, see Section 5.3.

NOTES:

The **SYS_router_configure** is actually not a syscall, but a wrapper for the syscall **SYS_router_cfg**. The only difference is that the *status*, *port_in*, *lane_in*, *port_out* and *lane_out* are packed in one integer.

SYS_conf_lane_in

SYNOPSIS:

void **SYS_conf_lane_in**(*int* lane, *int* config)

DESCRIPTION:

The *PIPE_montium_in* pipes can be configured for different data formats. Currently, there are four formats defined. For more information, see Section 5.3.

SYS_conf_lane_out

SYNOPSIS:

void **SYS_conf_lane_out**(*int* lane, *int* config)

DESCRIPTION:

The *PIPE_montium_out* pipes can be configured for different data formats. Currently, there are four formats defined. For more information, see Section 5.3.

debugging

SYS_strprint

SYNOPSIS:

void **SYS_strprint**(*const char* * msg)

DESCRIPTION:

The **SYS_strprint** prints a message to the device that is configured as "stdout". Normally, this is the serial port.

SYS_printf

SYNOPSIS:

void **SYS_printf**(*const char* * format, ...)

DESCRIPTION:

The **SYS_printf** is a wrapper for **SYS_strprint**, which prints a message with format *format*. There are only three special format characters defined:

%x inserts a 8-character hexadecimal representation of the next argument.

%d inserts a 1 to 11-character decimal representation of the next argument.

%s inserts the next argument as a literal string.

Other characters are copied without interpretation.

Appendix D

Example Code

```
#include <config.h>
#include <scheduler/task.h>
#include <lib/swi.h>

/* the pipe structure */
struct pipe *pipe;

// implementation of task 1
void task_1( void )
{
  // current_time is incremented on every call to task_1
  static int current_time = 0;
  current_time++;

  // write the current time to the pipe
  SYS_write_nb(pipe, &current_time, sizeof(int));
}

// implementation of task 2
void task_2( void )
{
  // read the current time from the pipe
  int time;
  SYS_read_nb(pipe, &time, sizeof(int));

  // and print it on the console
  SYS_printf("%d seconds have elapsed..\n", time);
}

static const struct task_specification task_1_spec = {
  .entry          = &task_1,
  .stack_size     = 256,
  .deadline       = 0x2000, // 250 ms
  .period         = 0x8000, // release the task every second
  .cputime        = 0x1000, // 125 ms
  .name           = "task_1",
  .flags          = TASK_FLAG_RT | TASK_FLAG_IDLING,
```

```
                        // task is a real-time task, start waiting

40    // shared resources: none
      .resource_sh      = NULL,
      .resource_ex      = NULL,
    };

45  static const struct task_specification task_2_spec = {
      .entry            = &task_2,
      .stack_size       = 256,
      .deadline         = 0x2000, // 250 ms
      .period           = 0x8000, // 1 second
50    .cputime          = 0x1000, // 125 ms
      .name             = "task_2",
      .flags            = TASK_FLAG_RT | TASK_FLAG_IDLING,
                            // task is a real-time task, start waiting

55    // shared resources: none
      .resource_sh      = NULL,
      .resource_ex      = NULL,
    };

60  void
    __start__( void )
    {
      /* create tasks */
      struct task *task_1 = SYS_task_create( &task_1_spec );
65    struct task *task_2 = SYS_task_create( &task_2_spec );

      /* create pipe */
      pipe = SYS_pipe_create(7);
      // create a 7 byte pipe. the size should at least
70    // be 4 bytes. it doesn't have to be a multiple of 4.

      // set signal thresholds
      pipe->rd_threshold = 4; // signal when 4 bytes can be read
      pipe->wr_threshold = 4; // signal when 4 bytes can be written
75
      // notify task_1 when data can be written to the pipe
      SYS_signal_add_listener( &(pipe->sig_wr), task_1 );

      // notify task_2 when data can be read from the pipe
80    SYS_signal_add_listener( &(pipe->sig_rd), task_2 );

      /* add tasks to the scheduler */
      SYS_scheduler_add_task( task_1 );
      SYS_scheduler_add_task( task_2 );
85  }
```