Implementation of an Interrupt-Driven OSEK Operating System Kernel on an ARM Cortex-M3 Microcontroller

Studienarbeit im Fach Informatik

vorgelegt von

Rainer Müller

geboren am 14. Februar 1987 in Erlangen

Angefertigt am

Lehrstuhl für Informatik 4 – Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:

Dipl.-Inf. Wanja Hofer
Dr.-Ing. Daniel Lohmann
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

Beginn der Arbeit: 01.05.2011
Abgabe der Arbeit: 27.10.2011

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 27.10.2011

Zusammenfassung

Ein Betriebssystem unterscheidet gewöhnlich zwischen Threads, die von einem Scheduler in Software verwaltet werden, und Interrupthandler, die von der Hardware eingeplant und eingelastet werden. Diese Unterscheidung trägt Probleme für Echtzeitsysteme mit sich, da Interrupt Handler mit niedriger Priorität Threads mit hoher Priorität unterbrechen können. Das Sloth-Konzept stellt einen Weg vor, dieses Problem durch die Implementierung von beiden, Interrupthandlern und Threads, als Interrupt zu lösen. Dies beseitigt den Unterschied zwischen den beiden Arten von Kontrollflüssen und diese Vereinfachung erlaubt es, das Einplanen und Einlasten von Threads vom Unterbrechungssubsystem der Hardware erledigen zu lassen.

Im Rahmen dieser Arbeit wurde dieses Sloth-Konzept als interrupt-gesteuertes Betriebssystem auf Basis der OSEK-Spezifikation für den ARM Cortex-M3 Mikrocontroller implementiert. Dies beinhaltet auch die Untersuchung, wie das Sloth-Konzept auf der zur Verfügung stehenden Hardware umgesetzt werden kann und wie die Hardware-Komponenten genutzt werden müssen. Diese fertige Implementierung wird dann evaluiert und mit einem herkömmlichen System mit einem software-gestütztem Scheduler verglichen, um die positiven Effekte dieses Konzepts auf die Verwaltung von Threads zu bestätigen. Ebenso wird der Einfluss der Hardware-Architektur auf das Design und die Implementierung von Sloth untersucht.

Abstract

An operating system usually distinguishes between threads managed by a software scheduler and interrupt service routines, scheduled and dispatched by an interrupt controller. This paradigm has inherent problems for real-time systems as low-priority interrupt routines can interrupt high-priority threads. The Sloth concept proposes to overcome this issue by implementing both interrupt handlers and threads as interrupts, which are scheduled and dispatched by hardware. This eliminates the difference between the two types of control flows by introducing a unified abstraction. With this simplification, scheduling and dispatching of threads can be managed completely by the interrupt subsystem in hardware.

In the scope of this thesis, this Sloth concept was implemented as an interrupt-driven operating system conforming to the OSEK specification on the ARM Cortex-M3 microcontroller platform. This entails the investigation how the Sloth concept can be implemented with the provided hardware functionality and how the hardware components need to be utilized. This finished implementation is evaluated and compared to another operating system with a software-based scheduler in order to confirm the positive effect of this concept on the performance of thread management. Additionally, this thesis examines the influences of the hardware architecture on the design and implementation of Sloth.

Contents

Chapter 1

Introduction

The scheduler is a key component in any operating system, as it is responsible for the management of different control flows. A scheduler usually switches between several threads synchronously activated by software. Additionally, interrupt service routines (ISRs) interrupt the CPU at any time asynchronously when signaled by the hardware. Using these two mechanisms together establishes inherently dual priority spaces, in which ISRs always have a higher priority than threads as they are running at the lowest hardware priority. Thus, an ISR meant to have a low priority can preempt a high-priority thread. This issue is known as *rate-monotonic priority inversion* [1]. The Sloth concept introduced in [2, 3] proposes to solve this problem by removing the distinction between threads and interrupts. By implementing all control flows in the system as interrupt handlers, the interrupt subsystem hardware can do the scheduling work. In traditional systems with a software scheduler, ISRs always have a higher priority as all threads. In Sloth, the unified priority space of threads and ISRs allows arbitrary distribution of priorities between them, without implying restrictions on the precedence of asynchronous over synchronous control flows.

1.1 The Sloth Concept

In Sloth, both types of control flows—threads and interrupt handlers—are implemented as interrupts. These threads running as ISRs are scheduled and dispatched by an interrupt controller in hardware, eliminating the need for a software scheduler completely. In this system, threads and interrupts share a single priority space managed by the hardware, avoiding the problem of rate-monotonic priority inversion as described above.

The implementation of the Sloth concept targets an event-driven embedded system, implementing the OSEK OS specification as an example. The offered API is the same as in traditional systems with a software-based scheduler by using this established standard; it is therefore trivial to port applications to run using the Sloth kernel. The description will stick to the terminology and system services of OSEK, although this concept can be applied in general to any event-driven real-time operating system.

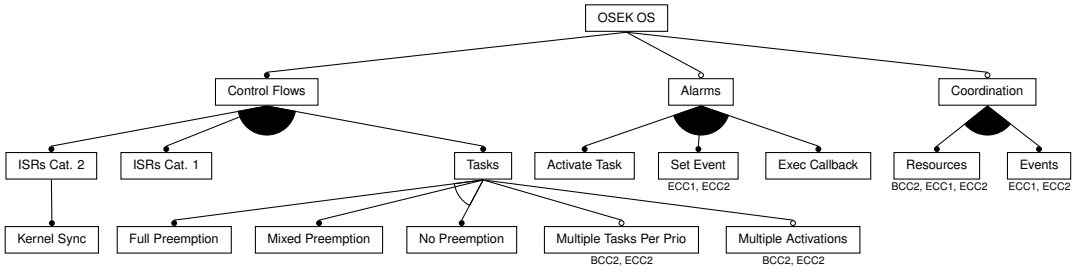

Figure 1.1: Feature diagram of the OSEK OS specification. Features mandatory in a conformance class other than BCC1 are marked with the corresponding classes below that feature. Feature types include mandatory features (filled circle), optional features (hollow circle), minimum-one feature sets (filled arc), and exactly-one feature sets (hollow arc) [2].

1.2 About OSEK

OSEK[1] is a standard for operating systems in the automotive industry. The specification for the OSEK OS [4] defines an interface to the operating system offering the necessary functionality for event-driven control systems. An overview of the provided features is given in Figure 1.1.

The offered system functionality includes control flow abstraction by use of *tasks* with different preemption policies that configure whether a higher-priority task can preempt another currently executing task (full preemption) or not (no preemption). This can also be configured individually for each task (mixed preemption). Optionally, the OS stores multiple activation requests for tasks and more than one task can share the same priority. Interrupt service routines (ISRs) are dispatched by hardware and are partitioned into two groups. ISRs of category 1 are not allowed to use any system services, while category-2 ISRs can use them. Synchronization is possible by acquisition of resources implementing the OSEK priority ceiling protocol to avoid priority inversion. Tasks are available in two types, where *extended* tasks have the same functionality as *basic* tasks, but can also yield the CPU and wait until a certain event occurs, which is signaled by another task. Alarms can activate tasks or execute callbacks after a specific period of time.

In order to build a highly scalable and portable system, the OSEK OS specification defines multiple conformance classes as shown in Figure 1.1. This allows partial implementation of the specification. Each conformance class defines a minimum set of features which are determined by the type of tasks present in the application, support for multiple activations of the same task, and the number of tasks per priority. These features are chosen and configured statically; that is, the tasks, ISRs, and their priorities as well as the system's features are configured at compile time.

1.3 Sloth Overview

In a system implementing the Sloth concept, each task and ISR will be assigned to an interrupt source with the configured priority. ISRs are activated by the hardware system as usual, whereas for tasks, two methods of activation are possible. On the one hand, they can be started syn-

[1]German acronym for "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug", translates to "open systems and corresponding interfaces for automotive electronics"

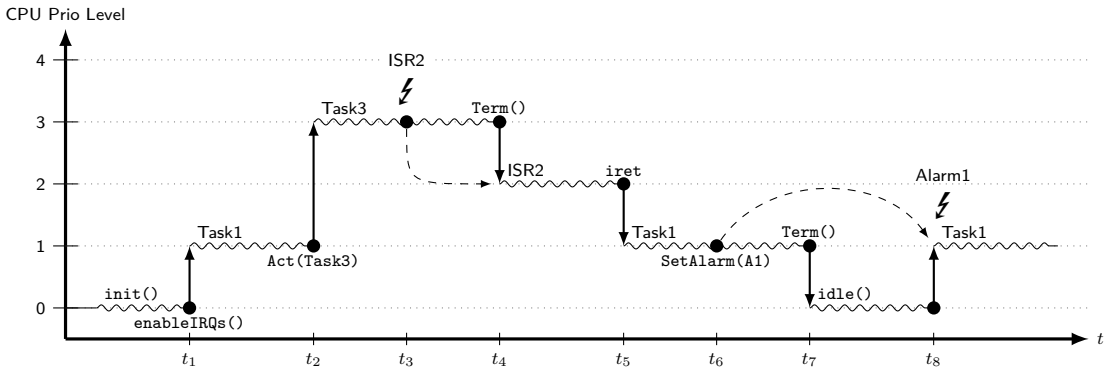

Figure 1.2: Example control flow in a SLOTH system. The execution of most system services leads to an implicit or explicit altering of the current CPU priority level, which then leads to an automatic and correct scheduling and dispatching of the control flows by the hardware. [2]

chronously by an OSEK system service, which leads to a software generated interrupt request. On the other hand, hardware devices can signal the corresponding interrupt asynchronously; for example, tasks triggered by an alarm are invoked by a timer system connected to the interrupt sources. Activation of a specific task results in triggering the interrupt source in both cases.

Subsequently, the interrupt controller needs to decide which of the interrupt sources with a pending request has the highest priority, which correspond to the assigned control flows—either tasks or ISRs. If the current execution priority of the CPU is less than the one of the determined interrupt source, the CPU needs to be interrupted by an interrupt request. The current execution priority does not always correspond to the executing control flow as it can be raised and lowered for synchronization purposes. If an preemption of the currently running control flow is possible, the task or ISR corresponding to this interrupt source will be dispatched. This scheduling implemented in hardware is responsible for the arbitration of the different priorities between tasks and ISRs. Rescheduling needs to take place every time an interrupt source is triggered, an interrupt handler returns, or masking of interrupts is enabled or disabled. Termination of a task matches the return from an interrupt handler, which will issue a new arbitration in the interrupt controller, determining the next interrupt to be handled.

Figure 1.2 shows an example control flow in a system implementing the SLOTH concept. In this configuration, Task1, ISR2, and Task3 have the priorities 1, 2, and 3, respectively. After initialization of the system, the auto-started Task1 starts running with priority 1 at t_1. In its execution at t_2, Task1 activates the higher-priority Task3, which is immediately dispatched and starts running with the CPU priority level being raised to 3; preempting the previously running Task1. At the time t_3, a hardware device signals an interrupt request for ISR2. However, as the current execution priority is 3, the execution of ISR2 has to be delayed until Task3 terminates at t_4. At this point, ISR2 starts executing until it returns from the interrupt at t_5 and the preempted Task1 continues running. When Task1 terminates at t_7, no other control flow is currently pending. Thus, the system is running an idle function at the lowest priority level waiting for interrupts. Here, at t_8, Task1 is again activated by Alarm1 that was previously set up at t_6.

1.4 Requirements on the Interrupt Controller

SLOTH's goal is to implement a very concise kernel utilizing hardware components—especially the interrupt controller. Using hardware for scheduling is supposed to improve the performance of context switches as compared to software-based schedulers. For this approach, SLOTH has requirements on the interrupt controller of the platform that define whether the system can be implemented:

- The interrupt subsystem must provide different priority levels for each task and ISR in the system.
- The interrupt subsystem must support software-generated interrupts, allowing to trigger individual interrupts in a synchronous manner.
- The platform must provide a way to raise and lower the current execution priority of the CPU as a synchronization mechanism.

Many modern platforms fulfill these requirements. The reference implementation of SLOTH was achieved on the Infineon TriCore, while this thesis details the implementation on the ARM Cortex-M3.

1.5 The ARM Cortex-M3

The SLOTH concept is not bound to a specific hardware architecture, but can be put into practice on any platform that fulfills the requirements discussed in Section 1.4. The reference implementation of the SLOTH concept was achieved on the Infineon TriCore platform [5], which is commonly used in the automotive industry. In this thesis, the SLOTH design was ported to the ARM Cortex-M3 microcontroller [6], to see where differences in the hardware design will demand a different approach in the implementation, which will then be evaluated in the comparison with the reference implementation in Section 4.2.

1.5.1 Architecture Overview

In 2004, ARM introduced the Cortex-M3 microcontroller, which targets a broad range of embedded system applications especially in the automotive segment [7]. The Cortex-M3 was the first processor of the ARMv7-M architecture profile, which is meant for small sized embedded systems optimized for deterministic operation and low-cost applications. The platform is designed as a 32-bit RISC system with a Harvard architecture, on which the processor operates in little endian byte order for both data and instruction accesses [8, 9].

The Cortex-M3 implements the Thumb-2 instruction set, which consists of mostly 16-bit instructions with a few additional 32-bit wide instructions. This simultaneous use of instructions with differing length achieves a higher code density while barely affecting performance. All instructions are aligned on a halfword boundary. The Thumb-2 instruction set is designed to support interworking with the older purely 32-bit ARM instruction set, in which all instructions have to be aligned to word boundary. To allow switching between the two instruction sets at all branch operations, the current execution mode is being kept as part of the program counter in bit 0, which is normally not being used due to alignment. On ARMv7-M, this bit must always be set to '1' to indicate use of the Thumb instruction mode, as only this is supported. This is

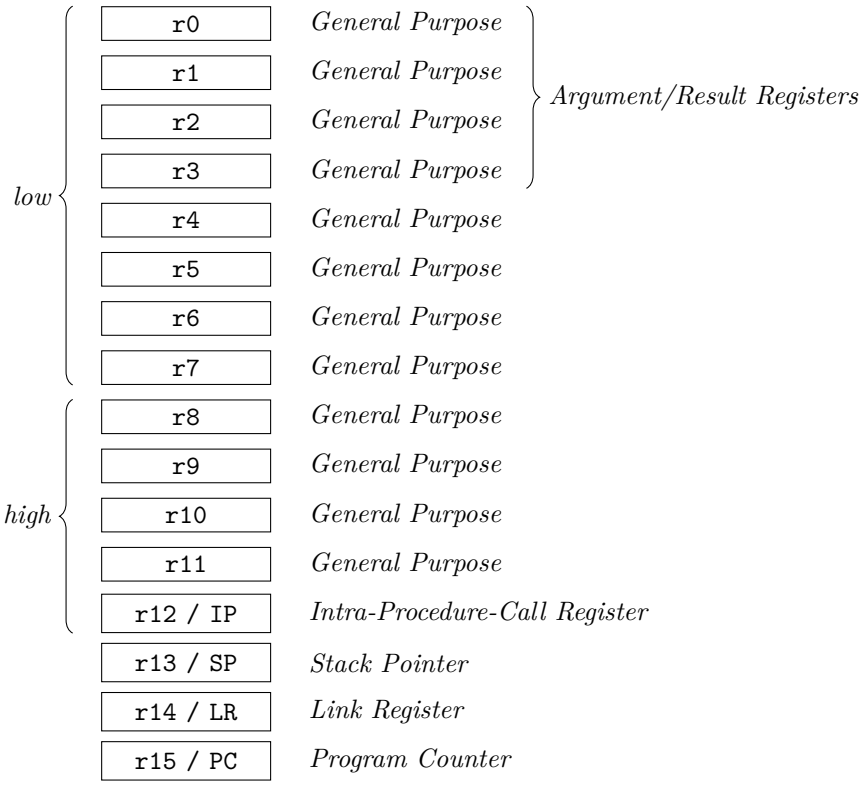

Figure 1.3: Registers in the ARMv7-M architecture.

usually hidden from the programmer as the compiler takes care of this, but can be important for low-level development, for example while examining addresses on the stack.

ARMv7 specifies 16 registers, each of which is 32 bits wide. They are arranged into 12 general-purpose and 4 special registers as shown in Figure 1.3. All instructions specifying a general-purpose register can access the *low* registers `r0`–`r7`. The *high* registers `r8`–`r12` are accessible from all 32-bit instructions, but not from all of the 16-bit instructions.

Register `r12` is used as a temporary workspace for address calculations during subroutine calls; register `r13` is used as the stack pointer (`SP`). Stack pointer values have to be aligned to a 4-byte boundary. The register is banked into `SP_main` and `SP_process` to allow easy stack switches for exception handlers and processes. Register `r14` is used as a link register (`LR`), which holds the return address during a procedure call. The return from a function is initiated by loading the `LR` value into the program counter `PC`, which is the special register `r15`. Additionally, the special program status register (`PSR`) holds current execution state like flags, which exception is currently being handled, and auxiliary state for load multiple and store multiple instructions.

The processor has two operation modes: *Thread mode* and *Handler mode*. Thread mode is the default operation mode after startup, which can run either in privileged or unprivileged level to limit access to some resources and system configuration. The Handler mode is activated while handling an exception and is always executed in privileged access level.

1.5.2 The Atmel SAM3U

For this thesis, an Atmel SAM3U4E evaluation kit from the SAM3U series was used [10], which includes a Cortex-M3 revision 2.0 running at up to 96 MHz with 52 kB of SRAM and 256 kB internal flash memory. On this particular SAM3U board, 30 different external interrupt sources and 16 priority levels are available. With the interrupt controller component in the Cortex-M3, this platform fulfills the requirements listed above in Section 1.4.

1.6 Outline of This Thesis

The following Chapter 2 gives a detailed overview of the features offered by the Nested Vectored Interrupt Controller as part of the ARM Cortex-M3, which will be the main hardware component utilized to implement the Sloth kernel. The design and implementation of Sloth for the Cortex-M3 are presented in Chapter 3, which is then evaluated in comparison with another OSEK implementation in Chapter 4. This chapter also highlights the differences between the system developed in the scope of this thesis and the reference implementation of Sloth on the Infineon TriCore. Chapter 5 concludes this thesis with with a summary of the results and an outlook of ideas for future work.

Chapter 2

The Nested Vectored Interrupt Controller

In order to implement the SLOTH concept on a platform, the hardware must fulfil the requirements as stated above in Section 1.4. The ARM Cortex-M3 microcontroller—the target platform for a SLOTH implementation in the scope of this thesis—includes a tightly-coupled interrupt controller called NVIC (*Nested Vectored Interrupt Controller*) [6, p. 6-1]; this integration of the interrupt controller into the core allows low-latency interrupt handling. The maximum interrupt latency amounts to 12 cycles, which describes the time between asserting the interrupt and executing the first instruction of the handler [6, p. 3-20].

2.1 Exceptions and Interrupts

The Cortex-M3 provides 16 system exceptions and allows up to 240 different external interrupt sources as shown in Figure 2.1. The Reset, NMI (*Non-Maskable Interrupt*) and HardFault exceptions have fixed priorities that cannot be changed. All other system exceptions and external interrupt sources can be assigned to one of up to 256 different priority levels. However, chip manufacturers can choose to only implement a fraction of these; for example, on the Atmel SAM3U4E used for this thesis, 16 different priority levels are available.

The ARM terminology refers to exceptions as running any sort of handler for both synchronous and asynchronous system events. The first 16 exceptions numbered 0 to 15 are reserved by the system. They are used for fault handlers (HardFault, MemManage, BusFault, UsageFault), debug functionality (Debug Monitor), supervisor calls (SVCall, PendSV) and for the system timer (SysTick). External interrupts start after that, so the external interrupt N will have an exception number of $16+N$. The NVIC locates the exception handlers using a static vector table, which is a list of address pointers to the entry points of the handlers.

A priority value of 0 denotes the highest configurable priority, higher values correspond to lower priorities. The maximum value is defined by the implementation. The priorities of Reset, NMI, and HardFault are fixed, so they always have a higher priority than all other exceptions. If multiple exceptions have the same priority and they are pending at the same time, the one with the lower exception number takes precedence.

2.2 Programming the NVIC

The components of the ARMv7-M architecture are programmed by manipulating memory-mapped registers. The registers for the NVIC are accessible in a special address range, called the Sys-

Exception number	Interrupt number	Name	Priority
1		Reset	−3
2		NMI	−2
3		HardFault	−1
4		MemManage	*configurable*
5		BusFault	*configurable*
6		UsageFault	*configurable*
7–10		*(reserved)*	
11		SVCall	*configurable*
12		Debug Monitor	*configurable*
13		*(reserved)*	
14		PendSV	*configurable*
15		SysTick	*configurable*
16	0	External Interrupt 0	*configurable*
17	1	External Interrupt 1	*configurable*
...	...	...	...
16+N	N	External Interrupt N	*configurable*

Figure 2.1: Exceptions provided by the Nested Vectored Interrupt Controller. While the priorities of Reset, NMI, and HardFault are fixed, all other priorities can be configured.

tem Control Space (SCS), in which internal system components are configured. Accesses in this memory region are strongly-ordered, which means the transactions will be performed in program order. Reading and writing these registers is limited to privileged execution level only. These registers control masking of individual interrupts, set and clear their pending state and configure their priorities. Each interrupt source can be triggered asynchronously from connected peripheral devices or with synchronous instructions from software.

The NVIC provides multiple registers as listed in Table 2.1, controlling interrupt masks and their pending state. A particular interrupt can be masked using the ISER (*Interrupt Set-Enable Register*) and ICER (*Interrupt Clear-Enable Register*) bit fields. These are organized in groups, so that each bit in a 32-bit register corresponds to one interrupt. There will be as many bit fields as required for the number of interrupt sources implemented by the hardware manufacturer. A single read or write operation can retrieve or manipulate the state of each interrupt in the specific group. Writing a 1 to the corresponding bit position enables respectively disables the interrupt, and reading the value returns the current state of the interrupts. A disabled interrupt can still be asserted as pending, but it will not be dispatched.

Additionally, the NVIC offers bit fields named ISPR (*Interrupt Set-Pending Register*) and ICPR (*Interrupt Clear-Pending Register*) to read and change the pending state of external interrupts. They follow the same write and read semantics as ISER and ICER, where reading the value returns the current pending state of the interrupts. Writing to ISPR allows software to trigger interrupts from software. Additionally, STIR (*Software Triggered Interrupt Register*) can be used to trigger interrupts in software without privileged access.

Priority levels are configured in the IPR (*Interrupt Priority Register*) bit fields. The priorities of interrupts can be set individually using 8-bit fields, which are organized in groups of four to

Register	Name	Function
ISER	**I**nterrupt **S**et-**E**nable **R**egister	writing bit n allows interrupt $\#n$ to be handled
ICER	**I**nterrupt **C**lear-**E**nable **R**egister	writing bit n prevents handling of interrupt $\#n$
ISPR	**I**nterrupt **S**et-**P**ending **R**egister	writing bit n marks interrupt $\#n$ as pending
ICPR	**I**nterrupt **C**lear-**P**ending **R**egister	writing bit n clears pending state of interrupt $\#n$
IPR	**I**nterrupt **P**riority **R**egister	byte n defines the priority of interrupt $\#n$

Table 2.1: Overview of the memory-mapped registers offered by the NVIC for control of interrupts. When the number of supported interrupts is large enough, the actual registers are split over multiple 32-bit bit fields organized in groups.

fill a 32-bit register. On systems not implementing the full 8 bits, the lower bits of the individual fields ignore writes and read as zero.

In addition to masking individual interrupts as described above, it is also possible to influence interrupt handling globally by setting `PRIMASK` (*priority mask*), `FAULTMASK`, and `BASEPRI` (*base priority*):

`BASEPRI` A register with up to 8 bits with the same width as implemented for the priority registers (IPR). This register sets the minimum required base priority for exceptions. An exception will only be processed immediately if its configured priority value is lower than the current `BASEPRI` value. Of course, it will not be processed if another exception handler with an equal or lower priority value is currently active. Remember that lower priority values correspond to higher exception priority.

The highest priority of all active exception handlers and the value of `BASEPRI` is called the *current execution priority*, as that is the actual value required for preemption.

`BASEPRI_MAX` Actually the same register as `BASEPRI`, but with a conditional write. The required base priority will only be raised and never lowered. Any write to this register trying to set a value higher (i.e., a lower priority level) than the current value will be ignored.

`PRIMASK` A 1-bit register which, when set, prevents the dispatching of any exception handler but not the NMI and HardFault. This is equivalent to raising the current execution priority to 0, the highest configurable value.

`FAULTMASK` A 1-bit register similar to `PRIMASK`, but disables dispatching of all exceptions including the HardFault and allows only the NMI. This has the same effect as raising the current execution priority to -1, the priority of the HardFault exception handler.

These global masking registers are not memory-mapped, but are accessed using instructions instead. The MRS and MSR instructions (*Move Register to Special* and *Move Special to Register*) read and write special register values, including the `BASEPRI` value from or to a general-purpose register. The additional conditon of `BASEPRI_MAX` can be specified by the encoding of the special register passed to the MSR instruction. The 1-bit registers `PRIMASK` and `FAULTMASK` are written using the CPSIE and CPSID instructions (*Change Processor State*), which enable or disable the specified mask.

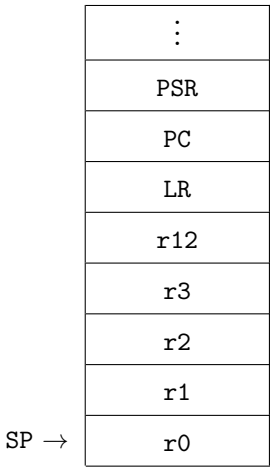

Figure 2.2: A stack frame as pushed onto the stack by the hardware at exception entry.

2.3 Exception Handling and Preemption

On exception entry, the NVIC automatically resets the pending bit of the dispatched interrupt and thus is able to assert this interrupt again as soon as it is handled. Additionally, it stores the flags, the return address—the value of the `PC` register at the time of interruption—, and scratch/caller-saved registers on the stack. These registers form the current state of program execution and will be used to resume where the interruption took place.

The pushed stack frame as shown in Figure 2.2 conforms to the AAPCS (*Procedure Call Standard for the ARM Architecture*, [11]). This accommodates C/C++ programmers as exception handlers can be written in C/C++ functions that are entered and returned using the calling convention without requiring special assembler instructions. The difference between a normal function call and interrupt handler entry is a special value in the `LR` register that will invoke the return from exception when written to the `PC`. This value encodes if the interrupted code ran in thread or handler mode and which of the banked stack point registers was in use.

The NVIC is capable of nesting interrupts automatically, which allows dispatching of a new interrupt handler while another interrupt handler is currently running. Therefore, interrupts will not be masked during execution of a handler. As only interrupts of a higher priority may interrupt the currently executing control flow, the dispatching of an interrupt handler raises the current execution priority to the priority of this interrupt. When another interrupt is triggered, its priority will be compared to the current execution priority. If the new interrupt has a higher priority, the current handler will be suspended and an exception entry for the new handler takes place. After this handler returns, the control flow of the initial handler will continue where it left off. Otherwise, if the priority of the new interrupt was lower, the initial handler runs to completion before the pending interrupt may be handled.

2.4 Summary

The NVIC as part of the Cortex-M3 fulfills the requirements for a Sloth implementation as defined in Section 1.4. It provides multiple priority levels, which can be assigned freely to any

interrupt source by setting the IPR registers. Also, the interrupt sources can be triggered in software by writing the bit pattern to ISPR, which are handled in the same way as if they would have been triggered from external peripherals. Finally, interrupts can be masked for synchronization purposes by raising the current execution priority with `BASEPRI`. Thus, the Sloth concept can be implemented on the ARM Cortex-M3 as presented in the next chapter.

Chapter 3

Design and Implementation

The SLOTH concept defines the goal of letting the hardware interrupt subsystem do the scheduling of control flows in an event-driven embedded system. The design of the SLOTH kernel for the ARM Cortex-M3 platform follows the design of the original implementation for the Infineon TriCore [2, 3]. New abstractions have been introduced for some of the functionalities where the different hardware requires a different approach. First, this chapter will discuss the utilization of the provided hardware systems; then, it explains the implementation of the OSEK system services in detail.

3.1 Sloth Implementation Overview

The implemented SLOTH system conforms to the classes BCC1 and ECC1 of the OSEK operating system specification as defined in Figure 1.1 in Section 1.2. Therefore the system provides support for tasks with statically configured priorities. These tasks are available as two types, of which *basic* tasks always run to completion and *extended* tasks can block and wait for an event. A synchronization mechanism exists in form of resources for mutual exclusion in critical sections. For periodic actions, alarms can activate tasks after a timer has elapsed.

3.1.1 Utilization of the SAM3U Hardware

This section gives an overview of how the hardware is used to implement the various system services of OSEK in order to achieve the goal of performing the scheduling by hardware.

The main part of the implementation is structured around the NVIC (*Nested Vectored Interrupt Controller*) provided with the ARM Cortex-M3. Each task is assigned to one of the interrupt sources. The external peripherals usually connected to these interrupts should not be used and need to be disabled in the power management to avoid any disturbance. As only the application specifies which of the external peripheral components will be in use, the mapping of tasks to interrupt sources is specified in the application configuration. Tasks also have a unique priority that will be configured in the NVIC. While in OSEK higher priority values correspond to higher priority, the NVIC defines 0 as the highest priority and higher values correspond to lower priorities. The application configuration always uses the definition of OSEK and maps the specified priorities to the hardware definition where required.

The basic task management is quite simple on the Cortex-M3. Synchronous task activation from software boils down to triggering the corresponding interrupt. This only requires a simple

one bit modification in the ISPR to pend the interrupt. If the activated task has a higher priority than the current execution priority, it will preempt the currently running task. At any time, a task can terminate by executing a return from interrupt. The NVIC already supports nesting of interrupt handlers on its own and thus stores the scratch context automatically before entering the interrupt handler. However, due to the possibility of terminating a task from subroutine level, a prologue and epilogue model is required to ensure correct behavior in all cases as described in Section 3.2.

Task blocking for event support is implemented by saving the context of the executing interrupt handler, disabling the interrupt source in the register ICER in the NVIC, and a return from the interrupt. The stored context will be used to resume the operation later when the task is unblocked by another task or ISR. This results in setting the interrupt source to pending and enabling the interrupt source again. The interrupt controller is then responsible for dispatching the blocked interrupt handler, which will run the task prologue restoring the previous context (see Section 3.3).

Resources are OSEK's terminology for mutex synchronization objects, which are used to protect critical sections and prevent concurrent access to shared data and peripherals by using a stack-based priority ceiling protocol. The Sloth implementation uses the functionality of the `BASEPRI` register to boost the current execution priority while the resource is held. As multiple resources can be acquired in a nested manner, a resource stack keeps track of the previous priorities. Due to the static system configuration, the size of this stack is limited to the maximum amount of resources used in the system and can be computed at compile time. The implementation of resources is detailed in Section 3.4.

To take action after a specific amount of time has elapsed, OSEK offers alarms, which can either activate a task or run custom callback functions provided by the application. An alarm that is configured to activate a task can do so by triggering the interrupt source directly in Sloth. The hardware timer counters of the Atmel SAM3U board are used to implement this system service. As these timer counters are connected to the interrupt subsystem, they match the Sloth concept of using functionality provided by the hardware (see Section 3.5).

At startup of the Sloth system, after peripherals are initialized and the initial stack is set up, the interrupts are configured according to the application configuration. This boot process sets the priority for each of the interrupt sources in the NVIC and also enables them if they have been assigned to a task. The interrupt vector table is loaded and tasks marked to be auto-started in the configuration are set to the pending state. As the system starts with interrupts disabled, these will not be dispatched until the initialization is completed and interrupts are allowed to be handled.

3.1.2 System Configuration

The system is statically configured at build time. While customizing the system according to the configuration, system generation is allowed to include or exclude features depending on whether they are used or not. This modular approach makes it possible to produce small operating system kernels that are tailored to the need of the application. The resulting binary is reduced in size by disabling features and, thus, consumes less memory, which is an advantage in embedded development where resource limitation is an important issue. Additionally, a smaller feature set results in performance improvements as some checks can be disregarded. Due to the use of inlining of the C compiler, calls to system services can often be replaced with the few instructions making up the system service.

Sloth consists of a platform-independent API that can be used by applications and internal

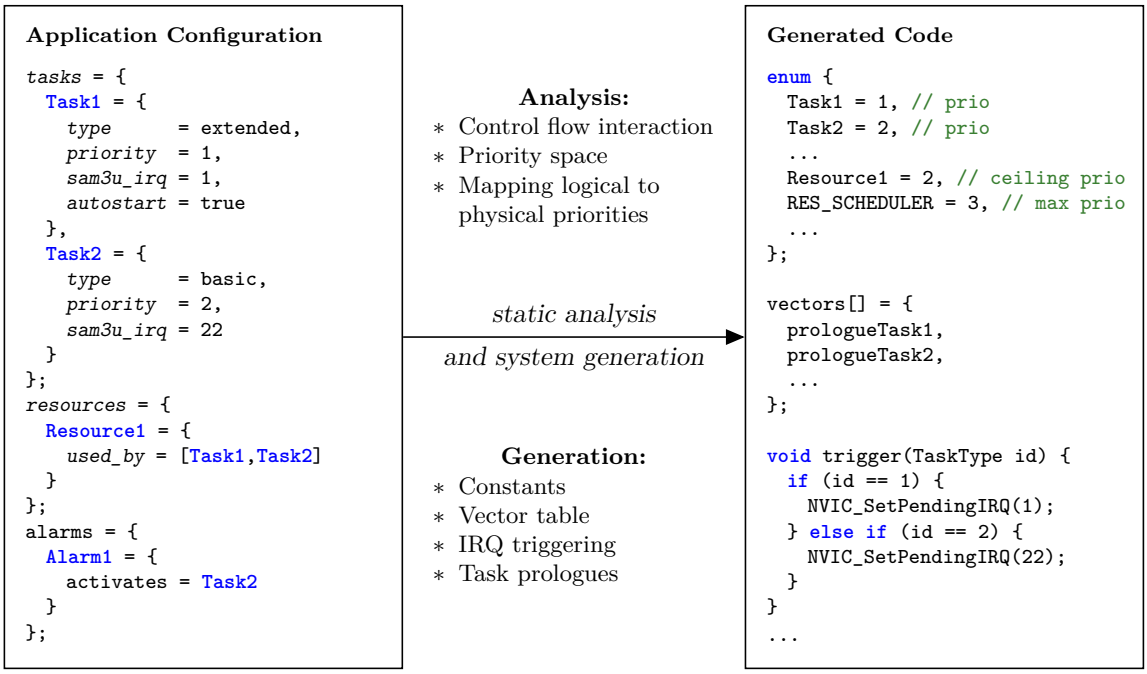

Figure 3.1: The example configuration of an application on the left is transformed into the code on the right by the static analyzer and system generator.

abstraction for functionality on different hardware platforms. Some features are implemented on the higher level and therefore do not require hardware specific handling.

The system generator written in Perl uses templates to produce header files according to the application configuration, for which an example is shown in Figure 3.1. The actual syntax almost corresponds to the OSEK Implementation Language (OIL), which makes it easy to port existing applications. The resulting files consist of the exception vector table, task stacks, and custom prologues for each task. The generator also calculates priorities of resources, creates functions to disable, enable or trigger interrupt sources and an initialization sequence for the interrupt controller and other peripherals used by the kernel, which also includes support for auto-starting tasks automatically at startup of the operating system.

3.2 Basic Tasks

Multiple control flows in complex software can be organized into tasks. These tasks provide the environment for executing functions. Usually a scheduler written in software is responsible for switching between them and thus defining the sequence of task execution. In Sloth, there is no software scheduler but instead the hardware is being used to determine task execution order.

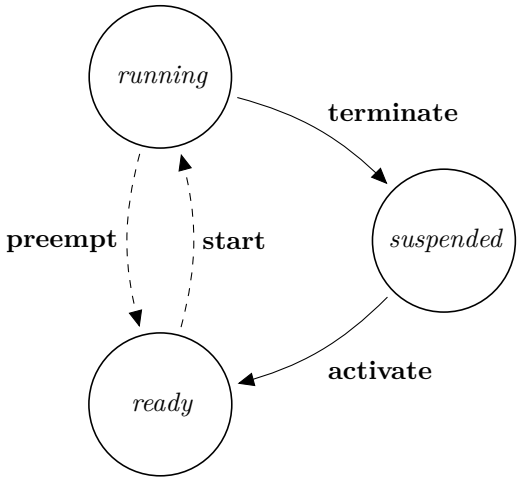

Figure 3.2: Basic task states with transitions defined by the OSEK specification [4, p. 18]. Transitions drawn dashed are implicitly handled by the scheduler, while solid transitions are invoked synchronously by software.

Overview

OSEK defines two different types of tasks, *basic tasks* and *extended tasks* [4, p .16]. This section is dedicated to the former, the latter with additional support for waiting for events are being discussed in Section 3.3 below.

As only one task can occupy the processor at any time, multiple tasks ready for execution may be competing for the processor at the same time. Additionally, tasks of higher priority can preempt tasks of lower priority. Thus, tasks transit between different states during the execution of an application. The operating system is responsible for switching between the tasks, while saving and restoring task context as needed in these transitions. The OSEK specification describes different policies that will change the preemption of tasks: *full-preemptive*, *non-preemptive*, and *mixed-preemptive*. The following sections focus on full-preemptive systems, which means that the operating system is allowed to reschedule the currently running task immediately as soon as synchronous or asynchronous task activations demand it. Data access shared with other tasks therefore needs to be synchronized using resources as described in Section 3.4.

Figure 3.2 depicts the different states and possible transitions for basic tasks. The currently executing task on the processor is in the *running* state, which can only be occupied by one task at a time. Tasks waiting for execution are in the *ready* state. Scheduling decides which task will change from the *ready* into the *running* state based on the configured priorities. The tasks of the basic type adhere to a run-to-completion paradigm commonly used in event-driven systems, which matches the run-to-completion execution model of interrupts, making them the perfect target for the Sloth concept. A task may only be moved back from *running* into *ready* when a higher-priority task or an ISR (*Interrupt Service Routine*) takes precedence. In the *suspended* state, a task has not been activated yet or terminated itself. A task will not be considered for execution in this state.

As basic task execution will only occur strictly nested, it is possible to share the same stack

for all basic tasks in the implementation.

System Services

Within an application, a task is declared using a special macro:

`TASK(`*`TaskID`*`) { ... }`

The *TaskID* will be used both in the configuration and system services to refer to this particular task.

The OSEK API models the transitions described above to the corresponding system services:

`ActivateTask(TaskType `*`TaskID`*`)`

`ActivateTask()` changes the state of the task corresponding to *TaskID* from *suspended* to *ready*. The operating system is responsible for starting the task as soon as possible according to task priorities and scheduling policy. A system may optionally support *multiple activations* in conformance class BCC2 or ECC2. In this case, the activation request will be recorded to be carried out later if the task is currently not suspended. Without support for multiple activations, a call will be ignored if the task is not in the *suspended* state. The configuration defines whether an application needs multiple activations or not.

In the Sloth implementation, the `ActivateTask()` function simply triggers the corresponding interrupt source in the ISPR register of the NVIC (see Section 2.2). An interrupt request will be generated by the hardware if the priority of the requested interrupt is higher than the current execution priority. Activating a task with a lower priority will only mark the interrupt source as pending and execution will continue with the calling task.

The implementation has to ensure that task activation happens synchronously. After activation of a higher priority task, preemption has to take place immediately and none of the next instructions of the calling task may be executed. As the ISPR register is mapped into the SCS (*System Control Space*), side-effects of the changes will take place immediately when the write access completes. An additional DSB (*Data Synchronization Barrier*) will be added to guarantee the access has always completed before proceeding. The ARMv7-M Architecture Reference Manual ([8, p. A3-119]) suggests to use ISB (*Instruction Synchronization Barrier*) to invoke a re-fetch of instructions already in the pipeline. However, the exception entry and return will flush the pipeline anyway, having the same effect as an ISB instruction here. If the activated task has a lower priority, no preemption is caused and the execution of the currently running task can just go on.

`TerminateTask(void)`

`TerminateTask()` changes the state of the currently *running* task to *suspended*, ending the execution of this task. There is no way to terminate another task; only the currently executing task can terminate itself. If this task is activated again later, execution will start at the first instruction. Ending a task without a call to either `TerminateTask()` or `ChainTask()` (see below) is not supported and may result in undefined behavior.

OSEK allows `TerminateTask()` to be called from a subroutine level of the task. As the Cortex-M3 only saves parts of the full register context on the stack automatically (see Section 2.1) and function calls decrease the stack pointer, this situation needs special handling. The solution

```
void prologueTask1(void)
{
    asm volatile ("push {%0, r4-r11, LR}" : : "r" (currentTask));
    returnSP[1] = currentStackPointer;
    currentTask = 1;
    asm volatile ("b functionTask1");
}
```

Figure 3.3: Implementation of the generated task prologue, here as an example for a task with the ID 1.

```
inline void __attribute__((noreturn)) epilogueTask(void)
{
    currentStackPointer = returnSP[currentTask];
    asm volatile ("pop {%0, r4-r11, LR}" : "=r" (currentTask));
    asm volatile ("bx LR");
}
```

Figure 3.4: Implementation of the task epilogue in a basic task system. All tasks use the same epilogue as the current task ID has to be determined at runtime.

here is to save the remaining registers and the stack pointer at the entry of the interrupt handler and restore that value when terminating the task. Usually the compiler would be responsible for saving and restoring non-scratch registers when they are used. However, in this case, the compiler-generated function epilogues need to be skipped to allow a premature exception return from subroutine level. Thus, in SLOTH, each task has an individual prologue prepended that saves the register values of `r4-r11` and `LR` on the stack (refer to Figure 3.3). Additionally, it stores the stack pointer value in a global array at the index of the task ID. A global variable `currentTask` is set to the current task ID in order to identify the currently running task later. Afterwards, a branch instruction to the actual task function starts the execution of the application code.

A call to `TerminateTask()` will run the epilogue of the task, which retrieves the saved stack pointer from the global array based on the `currentTask` variable. From this stack location, the register values saved in the prologue will be restored. Finally, it commences the usual exception return sequence by moving the special value in the `LR` register to the `PC`. The implementation can be seen in Figure 3.4. The attribute *noreturn* informs the compiler that a call to this function ends the current control flow, which will be used for optimization purposes.

`ChainTask(TaskType` *`TaskID`*`)`

`ChainTask()` will *terminate* the calling task. After the termination, the task corresponding to *TaskID* will be *activated*. The succeeding task is allowed to be identical to the calling task.

The implementation of `ChainTask()` is affected by the same problem as `TerminateTask()` as it can be called from a subroutine level. Furthermore, the activation of the succeeding task must not be performed before termination of the calling task. The defined point of the task switch should be the end of the task, which corresponds to the return from interrupt in SLOTH. Therefore, synchronization is required to ensure the correct order; the activation request needs to

```
inline void ChainTask(TaskType id)
{
    /* set FAULTMASK; raises current execution priority to -1 */
    asm volatile ("cpsid f");
    /* activate new task, sets pending bit of corresponding interrupt */
    ActivateTask(id);
    /* end this task */
    epilogueTask();
    /* implicit reset of FAULTMASK on return from interrupt */
}
```

Figure 3.5: Implementation of task chaining with use of `FAULTMASK`.

be delayed until the exception return has been executed, which requires special attention in the implementation for the Cortex-M3. In contrast to other microcontroller platforms, an exception return on the Cortex-M3 does not re-enable globally masked interrupts (`PRIMASK`). The `BASEPRI` value defining a minimum required priority level for preemption will not be affected either. This means that neither `PRIMASK`, nor `BASEPRI` can be applied for this purpose. However, while `PRIMASK` and `BASEPRI` values is not affected, the `FAULTMASK` will be reset on exception return. Although meant for a different purpose as described in Section 2.2, this functionality is used here to enforce the correct ordering of task execution.

Setting `FAULTMASK` raises the current execution priority over the configurable level and, therefore, this code section cannot be interrupted at all. The interrupt source corresponding to the succeeding task will be triggered next. Even if the new task has a higher priority, dispatching will be prevented by `FAULTMASK`. The calling task will then be terminated in the same way as `TerminateTask()` does. The exception return will implicitly disable the `FAULTMASK`. At this point, the hardware will decide which task or interrupt will be executed next according to the pending priorities.

3.3 Extended Tasks

Extended tasks in OSEK [4, p. 16] are an addition to the basic tasks introduced in the previous section. The extension on top of the functionality of basic tasks is the support of the OSEK event mechanism, which allows tasks to wait for an event and yield the processor until the event is set.

Overview

The event ID in conjunction with the owner, which can only be an extended task, identifies an individual event. An extended task as an owner of events is able to wait for one or more events at the same time. Only events owned by the calling task can be queried or cleared. Basic tasks and interrupt routines cannot own events and thus cannot wait for them, but they can signal events to their owners.

An extended task waiting for an event is transfered into the *waiting* state. This state is an addition to the basic task states as shown in Figure 3.6. In this state, it is no longer considered for scheduling and other tasks of lower priority can run in between. If at least one of the events the task is waiting for is signaled, the extended task will be moved from *waiting* to *ready*. If the

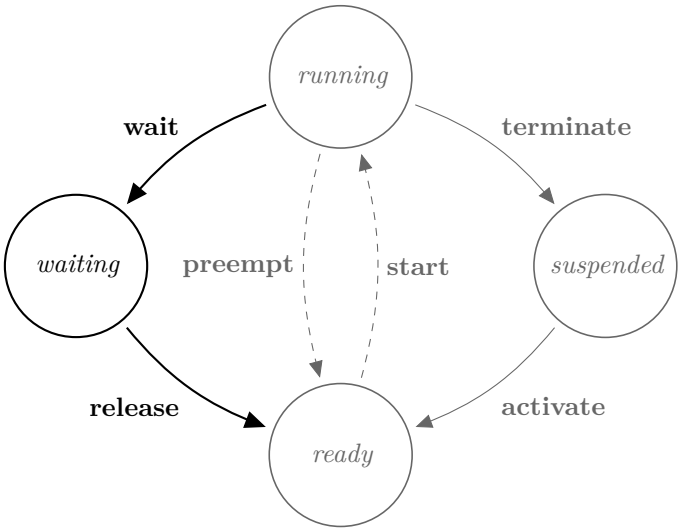

Figure 3.6: Extended task states as an extension to the basic task states as defined by the OSEK specification [4, p. 17]. This is an extension to the basic task model shown in Figure 3.2 on page 16.

event was signaled before the task tried to wait for it, this task remains in the *running* state. Signaling an event does not activate the task if it is currently in the *suspended* state. The event will be lost in this case.

While basic tasks can share the same stack as they can only preempt each other in a strictly stacked manner, extended tasks can block and wait for an event, during which the previous state of execution needs to be preserved while other tasks are executed. Therefore, it is necessary to assign separate stacks to each of the extended tasks.

System Services

SLOTH uses simple bit fields for the event mask of a task and the events a task is waiting for. The event mask holds the state of the individual events owned by this task; the events a task is waiting for are those that caused this task to be transfered into the *waiting* state. This *waiting* state is entered and left by blocking and unblocking extended tasks.

`GetEvent(TaskType` *`TaskID`*`, EventMaskRefType` *`MaskRef`*`)`

`GetEvent()` copies the state of all events owned by the extended task referenced by *TaskID* to the event mask pointed to by *MaskRef*. This function may be called from any task or interrupt service routine.

In the SLOTH implementation, this merely results in a simple check if the corresponding bits are set in the event mask of the calling task.

`ClearEvent(EventMaskType Mask)`

The service `ClearEvent()` clears the state of the events listed in *Mask* in the event mask of the calling task. Events always have to be cleared manually. This function may only be called from extended tasks that own the event specified by *Mask*.

The implementation for clearing an event is a simple bit field operation, removing the bits from the event mask of the calling task.

`SetEvent(TaskType TaskID, EventMaskType Mask)`

`SetEvent()` sets the event mask of the task corresponding to *TaskID* according to *Mask*. If the specified task was in the *waiting* state and waiting for at least one of the events denoted by *Mask*, it will be transfered into *ready* state. Other events of this task remain unchanged. This function may be called from any task or interrupt service routine.

The implementation of `SetEvent()` in SLOTH will be discussed below in conjunction with the `WaitEvent()` system service.

`WaitEvent(EventMaskType Mask)`

The `WaitEvent()` system service sets the mask of events to be waited for. This function may only be called from extended tasks and only events owned by this task may be specified in *Mask*.

In a system running basic tasks only, all control flows are strictly nested. The possibility of blocking and unblocking of extended tasks introduces more complexity as all other tasks—even those with a lower priority—can continue their execution while an extended task is in the *waiting* state. Interrupt handlers with a run-to-completion execution model are not designed to handle suspension and resumption; thus, to block a task, it has to be removed from scheduling completely. In SLOTH, this means that the interrupt controller needs to continue dispatching other interrupt handlers and their corresponding tasks while ignoring the blocked task. On the ARM Cortex-M3, the decision whether a pending interrupt handler can preempt the running control flow is based on the current execution priority. For this, the NVIC keeps track of all active exception handlers and takes them into account for calculation of the current execution priority. The Cortex-M3 will not dispatch any new handler with a lower priority assigned as long as the current execution priority is equal or higher. The priorities assigned to interrupts are used to determine if a new interrupt handler can be dispatched. However, the priority of a running interrupt cannot be lowered by the IPR value in the NVIC as the current execution priority is only calculated once at the time of dispatching of the interrupt. Therefore, to support the continuation model of extended tasks, SLOTH has to internally terminate extended tasks when they need to block to retain them from scheduling. In order to re-enter the interrupt handler later at the point of blocking, the full context of the task needs to be stored before it is terminated.

Blocking occurs whenever `WaitEvent()` is called and the event in question has not been set before. As shown in Figure 3.7, after checking the event mask and setting the mask of events to be waited for, the corresponding interrupt source of the calling task is disabled. This prevents dispatching of this interrupt handler as long as the task is in the *waiting* state. The interrupt is triggered immediately afterwards, so it fires as soon as the interrupt source is enabled again. To unblock a task when `SetEvent()` signals an event another task is waiting for, the respective interrupt source is enabled again.

To save the full context of the extended task on blocking, a return address is pushed onto the stack followed by all the register values. This return address points to the instruction right after the context saving and termination. This has to be the address right before the compiler-

```
inline void WaitEvent(EventMaskType mask)
{
    /* if at least one event is already set, return immediately */
    if ((eventMask[currentTask] & mask) != 0) {
        return;
    }

    /* no event was set; need to block until an event is signaled */
    eventsWaitingFor[currentTask] = mask;

    /* disable interrupt source for current task and set to pending */
    archDisableIRQSource(currentTask);
    archTriggerIRQ(currentTask);

    /* retrieve return address from label resumeTask */
    asm goto (
        "adr r0, %l[resumeTask]\n"
        "orr r0, #1\n"
        "push {r0}\n"
        : /* no output */
        : /* no input */
        : "r0"
        : resumeTask
    );
    /* store context on stack */
    asm volatile (
        "push {r0-r3, r12}\n"
        "mrs r0, PSR\n"
        "push {r0}\n"
        "push {r4-r11, lr}\n"
        : /* no output */
        : /* no input */
    );
    contextSP[currentTask] = currentStackPointer;

    /* end task, returns from exception */
    epilogueTask();

resumeTask:
    ; /* dummy to avoid warning "label at end of compound statement" */
}
```

Figure 3.7: Implementation of the function `WaitEvent()`

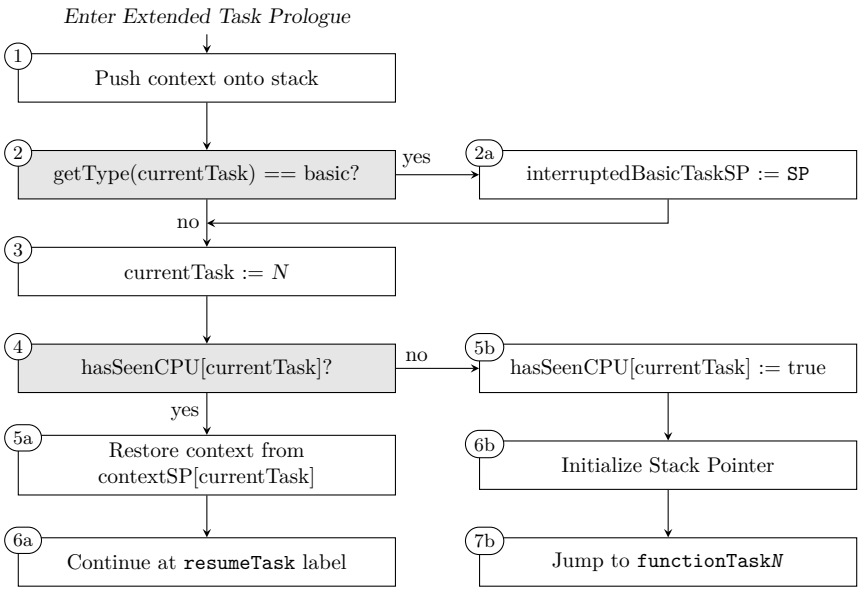

Figure 3.8: State diagram of the prologue for extended task N. As each prologue is generated individually, the number N will be replaced for the specific task.

generated function epilogue to restore any non-scratch registers used in the implementing function.

The SLOTH implementation uses the relatively new `asm goto` statement introduced with version 4.5 of GCC (*GNU Compiler Collection*) [12, 13]. This statement is an extension to the usual `asm` statement; it allows to retrieve the address of a C label and to be used in jump statements. This makes it possible to inline the implementing function. Using an `asm` label would have prevented inlining as the `asm` label can only be accessed in the global scope whereas here the address needs to be calculated each time the function will be inlined. The last bit of the address has to be set to 1 to indicate that the target location is assembled in the Thumb instruction set.

The current task context is pushed onto the stack and the stack pointer is saved to a global array indicating where the context can be found for the restore operation. Afterwards, the task is terminated using the epilogue for task termination which is described in detail in Section 3.2 above.

A matching prologue exists for extended tasks in the same way as for basic tasks, which is depicted in Figure 3.8. After saving the context of the interrupted task and setting the current task ID in steps 1–3, this prologue of an extended task additionally needs to check whether it has been activated or resumes from *waiting* state in step 4. If it has run before, the saved context needs to be restored by popping the values from the stack location back into the registers (step 5a). The return address pushed before will be written to the program counter in step 6a, completing the re-entry of the handler at the position after it was blocked. Otherwise, if the task did not run before, it is marked as running in step 5b. After initialization of the stack pointer in step 6b, the prologue jumps directly to the user task function in the last step 7b.

Stack switches are necessary in the prologue not only for extended tasks, but for basic tasks

as well. All basic tasks share the same stack as their invocation happens strictly nested, whereas each extended task uses its own stack. Extended tasks load their predefined stack pointer in the prologue in the initialization or implicitly set them by a restore of the saved context when returning from *waiting* state. A basic task preempting an extended task needs to return to the shared stack used by all basic tasks. Therefore, if the interrupted task was of the basic type, the prologue of an extended task saves the stack pointer of the basic task stack after saving its context. The prologue of the basic task will load this value and continue its operation on the basic task stack. When a basic task preempts another basic task, no action needs to be taken as they share the same stack.

3.4 Resource Management

Overview

The resource management in OSEK is responsible for coordination of concurrent accesses to shared resources [4, p. 29]. These resources could represent system components, memory ranges, or hardware peripherals. The operating system ensures that

- only one task can occupy a resource at a time,
- no deadlocks occur by use of resources, and
- priority inversion cannot occur.

To achieve these goals, OSEK prescribes a special kind of priority ceiling protocol. Resources are available in all OSEK conformance classes.

OSEK Priority Ceiling Protocol

The most common problems of synchronization methods like semaphores or mutexes are priority inversion and deadlocks. When using mutexes for synchronization, a task has to be blocked when the attempt to acquire a mutex was not successful as it was already occupied. Using a spin-lock is not possible in an event-driven uniprocessor system as taking it would immediately lead to an obvious deadlock. Additionally, priority inversion can occur when lower-priority tasks delay the execution of a higher priority task.

Figure 3.9 illustrates an example for an *unbounded priority inversion* in a full-preemptive system. Task T1 with the lowest priority acquires a mutex at t_1, after which T3 is activated at t_2 by an external event. As T3 has a higher priority, it preempts T1. During execution of T3, it tries to acquire the same mutex at t_4. As the mutex is already occupied by task T1, the task T3 has to block and return control to scheduling. The asynchronous activation of task T2 which took place in between at t_3 can now be handled as now other task with a higher priority than the one of T2 is ready. However, T3 cannot continue with its execution as long as T1 still occupies the mutex, which can only be unlocked when T1 is scheduled again. In this situation, T2 clearly delays the execution of the higher-priority task T3, although it is not even using the mutex.

To avoid such an unbounded priority inversion, the OSEK specification defines the OSEK Priority Ceiling Protocol. As an abstraction for synchronization, OSEK defines resources. When acquiring a resource, the current task's priority is raised to the resource ceiling priority and lowered to the original priority when released. This ceiling priority of a resource is the highest priority of all tasks that can acquire it. This value is computed at system generation and statically

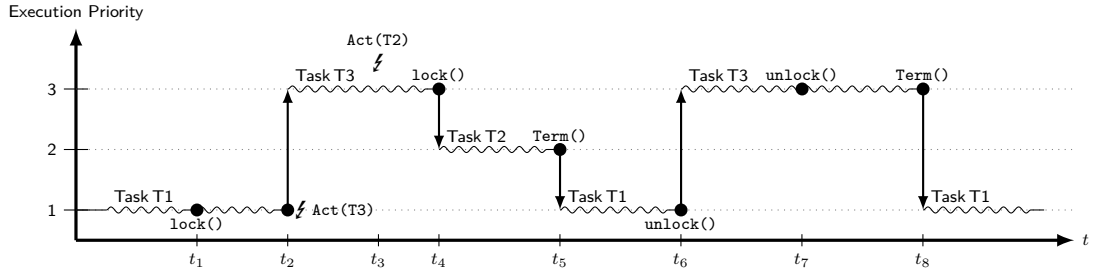

Figure 3.9: An example for an unbounded priority inversion problem using mutexes. The execution of the high-priority task T3 is delayed by task T2, although it has a lower priority.

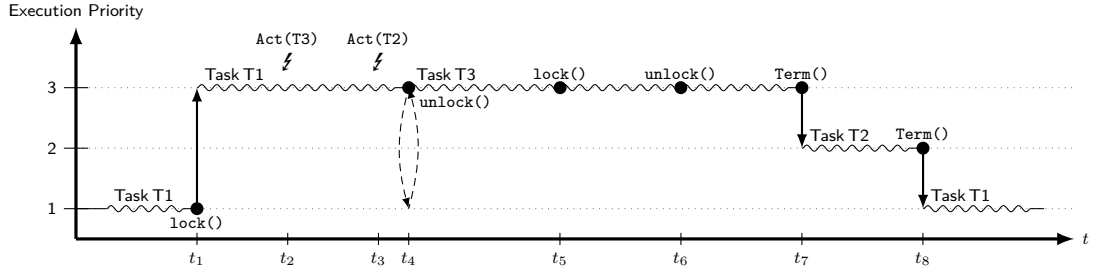

Figure 3.10: The solution to the unbounded priority inversion problem presented in Figure 3.9 using the OSEK priority ceiling protocol.

assigned to each resource. Tasks using the same resource therefore have a priority lower or equal to the resource's ceiling priority. Therefore, the ceiling priority required for a specific resource is the highest priority of all tasks accessing this resource.

The solution applied to the unbounded priority inversion problem using the OSEK priority ceiling protocol is shown in Figure 3.10. The priority of task T1 is raised to the ceiling priority when acquiring the resource at t_1 and falls down to the original value when releasing the resource in t_3. But at this point, the priority is not actually lowered again, as task T3 was marked pending by an external event at t_2 and is dispatched as soon as the execution priority allows at t_3. Hence, task T3 is only delayed for the time T1 occupies the resource in the critical section between t_1 and t_3. As no other task such as task T2 is able to run in between, the execution of T3 is not delayed any further.

This priority ceiling protocol ensures that no task can be preempted by another task accessing the same resource while the resource is occupied. Although this protocol still allows tasks with a priority higher than the one of the acquiring task, but lower or equal to the ceiling priority to be delayed, the duration of the delay is always limited to the time a resource is occupied by a lower-priority task.

```
inline void GetResource(ResourceType id)
{
    /* read current priority */
    TaskType localPrevPrio = archGetBASEPRI();
    /* lock the kernel; not interruptible after that */
    archSetBASEPRI(OSMAXPRIO);
    /* save the execution priority; increment resource stack pointer */
    resourceStack[resourceStackPointer] = localPrevPrio;
    resourceStackPointer++;
    /* check if new level is above previous level */
    TaskType newPrio = (id > localPrevPrio) ? id : localPrevPrio;
    /* set new priority level; implicitly unlocks the kernel */
    archSetBASEPRI(newPrio);
}
```

Figure 3.11: First implementation of the `GetResource()` system service, synchronizing the resource stack access by setting `BASEPRI` to OSMAXPRIO.

System Services

`GetResource(ResourceType` *`ResID`*`)`

`GetResource()` acquires the resource denoted by *ResID* to enter a critical section. Any critical section has to be left using `ReleaseResource()` before terminating the task. In case multiple resources will be acquired in the same task, the get and release operations need to be strictly nested.

The Sloth implementation uses the `BASEPRI` register of the Cortex-M3 to implement the priority ceiling protocol of OSEK. This register describes the minimum exception priority required for preemption as outlined in Section 2.2. The NVIC will only dispatch interrupt handlers with a priority higher than the current execution priority, which is usually the priority of the currently running interrupt handler. When multiple interrupt handlers are nested, this is always the one at the greatest nesting level which preempted the others. Using the `BASEPRI` register, the current exception priority can be raised to a higher level, which will prevent interrupt handlers—and the corresponding tasks—of a lower priority from dispatching. Changing the `BASEPRI` register can only raise the current execution priority over the priority of the currently executing exception handler. The initial value of `0` disables the masking of `BASEPRI` completely and only the priorities of the interrupts are taken into account.

In `GetResource()`, the current `BASEPRI` value is pushed onto a global resource stack; then, it is set to the new ceiling priority of the resource. Pushing the value onto the resource stack consists of multiple non-atomic operations. First, the resource stack pointer needs to be increased; then, the value is written into the reserved space. Thus, this access needs to be synchronized as another task of higher priority could preempt the currently running task in between and try to acquire a resource as well.

To make as much use of the provided hardware functionality as possible, two separate implementations using different synchronization mechanisms are provided. The configuration defines which of them will be used for the particular application. The first one shown in Figure 3.11 locks the kernel by increasing the current execution priority—the `BASEPRI` value—to the maximum pri-

```
inline void GetResource(ResourceType id)
{
    /* read current priority */
    TaskType localPrevPrio = archGetBASEPRI();
    /* lock the kernel; not interruptible after that */
    archDisableIRQs();
    /* save the execution priority; increment resource stack pointer */
    resourceStack[resourceStackPointer] = localPrevPrio;
    resourceStackPointer++;
    /* set new priority level if new value is greater */
    archSetBASEPRI_MAX(id);
    /* unlock the kernel */
    archEnableIRQs();
}
```

Figure 3.12: Second implementation of the `GetResource()` system service, synchronizing the resource stack access by disabling interrupts.

ority of all tasks and category-2 ISRs. The access to the `BASEPRI` register is encapsulated into wrapper functions `archGetBASEPRI` and `archSetBASEPRI`. The actual value is calculated during system generation as OSMAXPRIO.

In Sloth, tasks and category-2 ISRs, which are allowed to access system services, are not distinct as in traditional OSEK conforming implementations since they share the same priority space. ISRs of category 1, which cannot use system calls, will have priorities assigned higher than those of tasks and category-2 ISRs. Thus, raising the current execution priority to OSMAXPRIO will only lead to suspension of tasks and category-2 ISRs, while ISRs of category 1 with the higher priority can still be handled as usual.
The second implementation shown in Figure 3.12 utilizes the hardware by using `BASEPRI_MAX` to update the current execution priority. The `BASEPRI_MAX` register specification actually refers to the same register as `BASEPRI`, but has a conditional write. When writing a new value to `BASEPRI_MAX`, the hardware will compare it with the current value and applies the change only if the new value is greater than the current value. Otherwise, the write is ignored. This has the advantage that it is not necessary to read the value, compare it, and then write a new value as it would be done in software. Thus, as reading the value can be omitted, using `BASEPRI_MAX` should result in better performance for `GetResource()`.

However, the stack access still needs to be synchronized. To take advantage of `BASEPRI_MAX`, this cannot be done by raising the `BASEPRI` for locking purposes already. Therefore it is necessary to lock the kernel by disabling interrupts completely using `PRIMASK`, which is modified using the helper functions `archDisableIRQs` and `archEnableIRQs`. The drawback is that category-1 ISRs would be blocked as well, but only for the bounded time until the `GetResource()` system service is completed, which only takes a couple of cycles.

`ReleaseResource(ResourceType` *`ResID`*`)`

`ReleaseResource()` releases the resource denoted by *ResID* by restoring the previous value of `BASEPRI` from the global resource stack. The implementation is shown in Figure 3.13 and un-

```
inline void ReleaseResource(ResourceType id)
{
    /* lock the kernel; not interruptible after that */
    archSetBASEPRI(OSMAXPRIO);
    /* decrement resource stack pointer */
    resourceStackPointer--;
    /* restore the previous execution priority; implicitly unlocks the kernel */
    archSetBASEPRI(resourceStack[resourceStackPointer]);
}
```

Figure 3.13: Implementation of the `ReleaseResource()` system service.

like above for `GetResource()`, no alternative implementation is possible. `ReleaseResource()` is always synchronized using OSMAXPRIO with an implicit unlock by restoring the previous execution priority level. This leaves the critical section enclosed by `GetResource()` and `ReleaseResource()`.

`RES_SCHEDULER`

In an OSEK system, a task can protect itself against preemption by other tasks by acquiring the special resource `RES_SCHEDULER()`. This resource is automatically generated and accessible from all tasks. It is implemented as a normal resource following the priority ceiling protocol with a resource priority that is the same priority as the highest priority used for any task.

3.5 Alarms

Overview

Alarms in the OSEK operating system [4, p. 36] are used to run actions in either periodic or one-shot modes. They can be configured statically to activate specific tasks when a timer expires.

System Services

`SetRelAlarm(AlarmType` *`AlarmID`*`, TickType` *`increment`*`, TickType` *`cycle`*`)`

After *increment* ticks have elapsed, the assigned task will be activated. The *cycle* count can be used to set up periodic alarms, which will be set up again every time they expire. After the first invocation, the alarm will fire every *cycle* ticks. A value of `0` sets up a one-shot alarm.

The implementation uses the three timer counters included on the Atmel SAM3U board [10, p. 755]. These are organized around 16-bit counters driven by either the main clock modified by a scale factor or a slow clock running at constant 32,768 Hz. Each of them can be programmed to generate an individual interrupt when a specific counter value is reached. Depending on the application, the timer counters need to be configured to the correct clock sources. Using a different clock speed has an impact on the maximum *ticks* that can be specified for a `SetRelAlarm()` call. A task supposed to be activated by an alarm is mapped directly to the interrupt number of the corresponding timer counter in the configuration. For such tasks, a slightly different prologue will be generated as the external interrupt of the timer counter needs to be acknowledged. This

only adds one instruction and still works when activated manually using `ActivateTask()`. The `SetRelAlarm()` system service merely computes and sets the compare value for the corresponding timer counter derived from *AlarmID* and starts the timer. The counting will be done by the hardware until expiry of the timer activates the task by triggering its interrupt source.

The ARMv7-M architecture also provides a system timer called SysTick. This timer is capable of generating an interrupt after a specific tick count, but is limited to only one entry in the vector table. Thus, an implementation with support for multiple alarms would require a dispatcher in software to activate the corresponding tasks. To support this, additional data about the alarms would need to be kept in memory. Although using the SysTick would be more portable across different Cortex-M3 hardware, the timer counters as provided by the SAM3U board match the Sloth concept better.

3.6 Summary

The Sloth implementation detailed in this chapter implements OSEK system services for task management of basic and extended tasks, resource management, and alarms. Both task and resource management utilize the NVIC as part of the ARM Cortex-M3 for the implementation of interrupt-driven scheduling. The implementation of alarms uses the timer counters provided on the Atmel SAM3U board used for this thesis which are able to trigger interrupt sources after a given amount of time has elapsed.

Chapter 4

Evaluation

The Sloth implementation was evaluated on an Atmel SAM3U4E evaluation kit, which includes a Cortex-M3 revision 2.0. The board was configured to run at 48 MHz, which is the default clock speed in the startup code provided by Atmel. Measurements were taken in frequency-independent clock cycles to analyze the run time of selected scenarios in a preemptive operating system conforming to OSEK classes BCC1 and ECC1 in Section 4.1.

Additionally, due to the variability in hardware platforms, several aspects of the Sloth implementation for the ARM Cortex-M3 use a different approach as the reference implementation on the Infineon TriCore. Section 4.2 analyzes the differences in the systems and their implication on the system services.

Finally, the implemented feature set of OSEK is discussed in Section 4.3, explaining the current limitations of the system.

4.1 Performance Evaluation

To confirm the positive effects of the Sloth concept on the non-functional properties of an operating system, the performance of the Sloth implementation was compared to another OSEK-conforming implementation. For this purpose, Arctic Core was chosen, which is a multi-platform operating system developed as an open source project [14]. It implements the AUTOSAR specification, which is a superset of the OSEK specification using the same interface for system services. Although Arctic Core supports the ARM Cortex-M3 platform on several boards already, some small changes to the build system, startup code, and linker scripts against version 2.9.1 were necessary to add support for the Atmel SAM3U board used in this thesis.

4.1.1 Measurement Setup

The ARMv7-M architecture defines several components for debugging purposes. A debugger can be attached using a normal JTAG connector, which can read arbitrary registers and memory addresses, set breakpoints, and step through the code executed on the device. Additionally, the *Instrumentation Trace Macrocell* (ITM) and the optional *Embedded Trace Macrocell* (ETM) can generate timestamps and instruction traces, which can be read out using the Serial Wire Debug protocol. The Atmel SAM3U4E used for this thesis does not provide the ETM and only a JTAG adapter was available, which is not capable of reading the ITM data from the debug port.

Thus, a different approach using the *Data Watchpoint and Trace* unit (DWT) was used to take measurements of the implemented software.

The DWT provides, among other registers, a memory-mapped 32-bit cycle count register `CYCCNT`, which is incremented each clock cycle. This can be used to obtain the amount of clock cycles a specific code section needs for execution. After enabling the cycle counter in the DWT, the `CYCCNT` register can be read any time to get the current clock cycle count since the counter was reset.
A simple measurement using this approach would look like this:

```
uint32_t value;
DWT->CYCCNT = 0;
/* ... code to be measured ... */
value = DWT->CYCCNT;
```

As the DWT uses memory-mapped registers for control, the compiler-generated assembly code accessing `CYCCNT` needs to load its address first. For a small piece of code, this will be held in a register during execution of the examined code. However, for a larger section, the value will be calculated and loaded again. Depending on the surrounding code, an optimizing compiler might even try to merge the calculation with other statements, moving statements out of the examined section. To avoid falsified results or unsteady overhead to be added to the measured cycles, the register accesses were wrapped into small helper functions.

```
uint32_t value;
sam3uMeasureStart();
/* ... code to be measured ... */
value = sam3uMeasureStop();
```

This way, only two additional branch statements to `sam3uMeasureStart` and `sam3uMeasureStop` will be inserted in the code, respectively. Attributes advising the compiler not to inline these *start*/*stop* functions to ensure branch statements are also included. In the *stop* function, the measurement code automatically subtracts the overhead added by these calls, which is determined in an initial measurement at the beginning of the test program.

The execution time of a code section on the Cortex-M3 is usually predictable as long as no branch instructions are hit in the instruction stream and no interrupts occur in between. However, fetching instructions from the embedded flash controller may be delayed depending on the address and wait states applied. For example, a halfword-aligned 32-bit instruction will require a second fetch as all instruction fetches are word-wide. That is why the compiler was instructed to align the 32-bit branch instruction to the *start* function to a word boundary, which is meant to ensure the overhead introduced by the measurements is constant. Of course, depending on the measured code, the *stop* function might still hit a halfword boundary, which might lead to a single cycle added to the measurement. However, this is inherent to the measured code and cannot be avoided.

The flash controller on the SAM3U4E optimizes sequential read accesses using buffers, adding two wait states for the first instruction of a block by default. To simulate an environment as it would be used in real applications, this was not changed. Additionally, at each branch instruction, the 3-stage pipeline of the Cortex-M3 might need to be flushed if the branch prediction guessed wrong. Therefore, the initial measurement to calculate this overhead is performed with 8 NOP instructions. This number was chosen as the encoding of 16-bit NOP instructions is a multiple of four to mitigate the effects of misaligned instructions and is long enough to fill the pipeline of the Cortex-M3.

The measured values also need to be retrieved from the board. The host computer was connected over a serial interface using the *Universal Asynchronous Receiver/Transmitter* (UART) on the SAM3U hardware. The *start/stop* functions mentioned above were extended to transmit the measurement value right after it has been obtained to the host computer, where further processing may take place. All measurements were repeated multiple times, but as execution on the ARM Cortex-M3 is fully deterministic, this leads to zero deviation between all runs.

4.1.2 System Configuration

As both Sloth and Arctic Core support the OSEK system services, the same test applications were used on both systems. Both systems were compiled using the GNU Compiler Collection with the optimization level `-O3` and assertions disabled. The configuration was almost the same for both implementations. Whereas Sloth uses a shared stack for basic tasks, the Arctic Core implementation always uses distinct stacks for each of the tasks. This will have a slightly adverse influence on the performance of task switches in the Arctic Core system as they always involve switching stacks, although this would not be necessary in a BCC1 system with only basic tasks involved. Additionally, advanced features of Arctic Core were disabled by using the AUTOSAR *scalability class 1*, which is backwards compatible to OSEK.

For the Sloth implementation, an additional optimization regarding task switches was evaluated. Most system services need to know which task is currently active as they use this task ID as an offset in an array or for similar actions. Usually, this would involve loading the information from a memory location, which requires multiple bus accesses to load the address first and then the actual value. Thus, these accesses can be optimized by keeping the current task ID in a general-purpose register instead of a memory location. As all registers are normally under control of the compiler, a register has to be reserved globally and may not be used in any compiler-generated code. GCC allows to mark a register as *fixed* with the option `-ffixed-`*`reg`*, which removes it from the register allocator and generated code will never refer to it. As only *low* registers `r0-r7` can be addressed by all 16-bit instructions in the Thumb instruction set and using one of these would reduce the available registers for most instructions, the *high* register `r11` is chosen for this purpose. The downside of this approach is that all code has to be compiled with the appropriate compiler flag and, thus, linking against binary-distributed libraries is impossible.

Thus, the test cases were evaluated on two separate configurations for Sloth, where one stores the current task ID in a register and the other one uses conventional memory, and as the third competitor, the Arctic Core implementation. On all three systems, multiple scenarios were evaluated to show the performance of the implementation on task switching, synchronization with resources, and handling of events.

4.1.3 Test Scenarios

To cover as many scenarios as possible in the evaluation, three separate test applications were prepared for measurements. The first one uses basic tasks only, where the performance of task switches and resources using the OSEK priority ceiling protocol can be observed. The second one uses extended tasks only, where task blocking and the effects of the additionally required stack switches on the performance can be inspected. Finally, the last test application implements both basic and extended tasks, which allows to analyze the impact of extended tasks in the same system as basic tasks on their task switching performance and the scalability of the implemented system services. The selected test cases include all task-switching-related system services that stand to benefit from use of the hardware in the Sloth implementation.

Test Case	Sloth		ArcCore	Speed-Up
	register	memory		
A1 ActivateTask without dispatch	9	9	233	25.9
A2 ActivateTask with dispatch	40	56	507	12.7
A3 TerminateTask with dispatch	24	33	310	12.9
A4 ChainTask with dispatch	61	84	471	7.7
A5 GetResource a) synchr. using OSMAXPRIO	31	32	126	4.1
A5 GetResource b) synchr. using PRIMASK	21	22		6.0
A6 ReleaseResource without dispatch	29	29	185	6.4
A7 ReleaseResource with dispatch	63	80	461	7.3

Table 4.1: Performance evaluation of task switching and resources in a system with basic tasks only. The values specify the measured execution time in number of clock cycles. The speed-up is the comparison of the Sloth register variant with Arctic Core.

The latencies of all system services are measured from the point before the invoking statement until the action is completed. For instance, a preempting task activation is measured from the point before `ActivateTask()()` to the first application instruction in the activated task function, and a task termination is measured from the point before `TerminateTask()()` to the next instruction in the following task. If no preemption or task dispatching occurs, the measurement is ended at the next instruction right after the statement invoking the system service.

4.1.4 Basic-Task System

The results for task switching in a system with basic tasks only is presented in Table 4.1 as a comparison between the two Sloth configurations—with the current task ID in a register or in memory—and the Arctic Core system.

Sloth-Internal Evaluation Results

The numbers show clearly how Sloth can benefit from the optimization of keeping the current task ID in a general-purpose register for system services operating with this number. Although test case A1 is not influenced at all as activating a task without preemption does not involve the current task ID, test case A2 runs faster since the context saving in the prologue of the dispatched task can take advantage of the optimization. By using a register, the current task ID is stored altogether with other registers, whereas by using a memory location, a separate load and store sequence is required. Similar improvements can be observed in both task termination and task chaining in test cases A3 and A4. Overall, the register variant has a maximum speedup of 1.4 compared to using a memory location.

For resource acquisition in test case A5, the performance of the two implemented methods for synchronization as described in Section 3.4 were compared. Implementation a) raises the current execution priority in `BASEPRI` to OSMAXPRIO in order to synchronize accesses to the resource stack, whereas implementation b) takes advantage of the hardware by using `BASEPRI_MAX` for setting the new execution priority, which renders reading and comparing the value in software unnecessary. Therefore it has to implement a different way for synchronization by disabling interrupts completely using `PRIMASK`. The results show that both implementations are not influenced

very much by the register optimization, but implementation b) is 10 cycles faster than a) due to using the hardware to the full extent. However, the synchronization method using `PRIMASK` in b) also blocks ISRs of category-1, while synchronization in a) still allows them to be handled. Thus, the configuration of the system in this respect is subject to a trade-off decision by the application programmer.

The test cases A6 and A7 show the performance of releasing a resource, while in A7 a task dispatch is required and in A6 it is not. The `ReleaseResource()` system service itself does not gain performance from the optimization by using a register for the current task ID as that value is not used for this action. The difference in performance between A6 and A7 is similar to the difference between task activation with and without dispatching a task for both versions of Sloth, which is 31 and 34 cycles for the register variant and, when the current task ID is held in memory, 47 and 51 cycles. This shows that only the prologue of the dispatched tasks adds additional cycles to the operation.

Comparison with Arctic Core

In comparison with Arctic Core with a scheduler and dispatcher in software, a huge difference in performance can be noticed for all system services, totaling to between 95 and 224 additional clock cycles. Apart from its software-based scheduling, this arises from the fact that this system does not use a single stack for all basic tasks and thus has to perform a stack switch for each transition as the tests in the following sections will show. Additionally, Arctic Core does not use as many optimizations as Sloth does. For instance, Arctic Core compiles and links in multiple steps and thus, system services cannot be inlined into the application itself as they are in the Sloth implementation. Due to the software-based nature of Arctic Core, many cycles have to be spent for scheduling. This can be seen in the first four test cases which cover the task management for basic tasks, where even the slower configuration of Sloth that uses a memory location for the current task ID can achieve a minimum speed-up of 5.6, whereas the variant using a register even achieves a minimum speed-up of 7.7.

Summary

In summary, the difference between the versions with the current task ID in a register and memory has a maximum speed-up of 1.4. In absolute cycles, this ranges from 0 cycles for system services that do not need this information, up to 23 cycles for task chaining. This proves that reserving a register for the current task ID can be a viable optimization to be enabled for the Sloth implementation.

The performance comparison shows that the scheduling by hardware can achieve a speed-up ranging from 4.1 to 25.9 for a basic task system compared to the software-based scheduler of Arctic Core.

4.1.5 Extended-Task System

The second test application evaluates the performance of a system with extended tasks only. Here, every task switch requires a stack switch as well. The results of the measurements are shown in Table 4.2. As the performance gains of the variant using a register for the current task over using a memory location was already affirmed in the first test case, only the register variant of Sloth is compared to Arctic Core.

Test Case	Sloth	ArcCore	Speed-Up
B1 ActivateTask (extended) with dispatch	60	505	8.4
B2 WaitEvent with dispatch	96	325	3.3
B3 SetEvent with dispatch	87	407	4.7
B4 ClearEvent	9	48	5.3
B5 TerminateTask (extended) with dispatch	30	312	10.4
B6 ChainTask (extended) with dispatch (same task)	94	235	2.5

Table 4.2: Performance evaluation of task switching and blocking in a system with extended tasks only, which requires stack switches for all system services except B4. The values specify the measured execution time in number of clock cycles.

Sloth-Internal Evaluation Results

Although the blocking and unblocking of tasks does not fit to the run-to-completion model of interrupt handlers, the Sloth implementation achieves respectable performance. Task switches between extended tasks in B1 and B5 take a little bit longer than basic task switches in A2 and A3 with a difference ranging between 6–20 cycles compared to the register variant. This is because task switches for extended tasks involve stack switches and more conditional checks. The task termination in B5 requires less additional cycles with only 6 cycles compared to the register variant in A3 as the code path is very similar as the next test application will show in more detail. The additional cycles required for task activation with dispatch are added by the extended task prologue as it has to do stack switches and needs to check whether it resumes from waiting for an event or was dispatched without prior context.

The `ClearEvent()` system service in B4 takes the fewest time with 9 cycles as it only operates on a bit mask. The task blocking implemented in `WaitEvent()` as tested in B2 involves saving the full context state of the task in order to block the task and remove it from scheduling. This system service—including a following dispatch—takes 96 cycles in the Sloth implementation. The task unblocking caused by `SetEvent()`, which allows a blocked extended task to continue its operation, takes 87 cycles, which includes the extended task prologue which restores the previous context. Therefore, with 27 cycles more, this takes significantly longer than a task activation of 60 cycles in B1, where the prologue can directly jump to the user task function. Test case B6 of extended task chaining takes 33 cycles longer than basic task chaining in A4 which already includes the additional 6 cycles for task termination and additional 20 cycles for the extended task activation.

Comparison with Arctic Core

Again, all system services have an performance improvement over Arctic Core, while the speed-up ranging from 2.5 to 10.4 is not as high as it is for a basic task system. The extended task activation of Arctic Core in B1 is very similar to the basic task activation in A2 with a difference of only 2 cycles, which could also be caused by unaligned instruction accesses in the measurement as outlined in Section 4.1.1. The same is observed for task termination in B5 and A3, which also only differs in 2 cycles. However, this suggests that Arctic Core does not handle basic and extended tasks differently and especially requires the same overhead for scheduling and dispatching in software for both of them.

The lowest speed-up value was measured for task chaining in B6, which, as said, differences

Test Case	Task Type Transition	Stack Switch	Sloth	ArcCore	Speed-Up
C1 ActivateTask	Basic → Basic	no	42	505	12.0
C2 ActivateTask	Basic → Extended	yes	65	525	8.0
C3 ActivateTask	Extended → Basic	yes	47	525	11.2
C4 WaitEvent	Extended → Basic	yes	112	357	3.2
C5 SetEvent	Basic → Extended	yes	92	407	4.4
C6 TerminateTask	Basic → Basic	no	33	313	9.5
C7 TerminateTask	Extended → Basic	yes	33	312	9.5
C8 TerminateTask	Extended → Extended	yes (dispatch new task)	83	346	4.2
C9 ChainTask	Basic → Basic	no (dispatch same task)	74	233	3.1

Table 4.3: Performance evaluation of task switching and blocking in a mixed task system with both basic and extended tasks, which requires stack switches for transition involving extended tasks. The values specify the measured execution time in number of clock cycles.

from the basic task chaining in A4 as the former chains the same task again, where Arctic Core achieves a faster operation for B6 than for A4. Arctic Core probably optimizes the special case of chaining the same task, which does not need to switch stacks as it is safe to assume the currently terminated task will again be the task with the highest priority. Although Sloth does not apply any optimization for this case, it still outperforms the software-based scheduler with a speed-up of 2.5.

Summary

In the performance comparison with Arctic Core, the Sloth implementation achieves a speed-up ranging from 2.5 up to 10.4 for an extended-task system. This is not as high as for the first test application which only involves basic tasks. However, extended tasks do not match the run-to-completion model of interrupt handlers and this second test application shows that the Sloth implementation of task blocking again performs faster than a software-based scheduler.

4.1.6 Mixed Task System

The third test application examines the performance in a system with both basic and extended tasks; the results are shown in Table 4.3. The comparison is between Sloth, configured to use a register for the current task ID, and the Arctic Core implementation. The task transitions denote which task called the system service and which task is chosen to run next after the scheduler decision. The table also indicates required stack switches for transitions involving extended tasks.

Sloth-Internal Evaluation Results

In a mixed task system using the Sloth kernel, also the basic task switches are affected by a small performance penalty (for instance, 2–5 cycles in C1 and C2 when compared to A2 and B1), as checks whether the preempted task was extended—leading to a stack switch—are required. Similarly, task termination and task chaining execute additional instructions as they need to distinguish whether the calling task is basic or extended by checking the calling task ID.

Basic task activation from within an extended task in C3 only has a very small overhead of 5 cycles for the stack switch compared to the basic task activation from within a basic task in C1.

Task termination in test cases C6 and C7 has the same timing of 33 cycles as both test cases take the same code path in Sloth. At preemption, the context is always stored on the current stack and the stack pointer is saved to allow an exit of the task from subroutine level. Thus, restoring the stack pointer for a basic task is the same action as loading the stack pointer for an extended task. The results for C6 and C7 are similar to those in A3 and B5 with 3–9 additional cycles required. However, C8 is different, as in this test case, a new task is dispatched, whereas in C6 and C7, the termination returned to the activating, already running task. Therefore, the result for C8 also includes the execution time of about 56 cycles (derived from C2 and A1) required for the extended task prologue. Evaluation of the task chaining of two basic tasks in C9 is 20 cycles faster than task chaining of extended tasks in B6, which is about the same difference in amount of cycles added to task activation between basic and extended tasks due to the different prologues. Task blocking and unblocking in Sloth using the `WaitEvent()` and `SetEvent()` system services with task switches to basic and from basic tasks take additional 5–16 cycles as compared to a purely extended-task system.

Comparison with Arctic Core

The numbers for the Arctic Core system show that there is not much difference in task activation between basic and extended tasks in test cases C1, C2, and C3, as a separate stack is being used for each task, including basic tasks. For task termination, similar effects as in the Sloth system for C6 and C7 can be observed. Test case C8 dispatching a new task adds a less significant amount of additional cycles for Arctic Core. As mentioned above, the value obtained in A4 deviates from the other two test applications, as the test cases B6 and C9 chain the calling task again, while A4 dispatches another task.

The speed-up of 8.0–12.0 of Sloth for task activations in C1, C2, and C3 is similar compared to the speed-up of 8.4 and 12.7 observed in the test cases in A2 and B1. The same is true for the other test cases as well, which all achieve a little less speed-up than the test cases in the purely basic and extended test applications before. Nevertheless, the Sloth implementation achieves a speed-up in the range from 3.1 up to 12.0 in the third test application.

Summary

The third test application shows that the Sloth implementation is able to provide a system with both basic and extended tasks, in which the basic task switches only have a small overhead of up to 9 cycles.

As expected after the first two test applications, Sloth outperforms Arctic Core in the third test application once again with a speedup from 3.1 to 12.0.

4.1.7 Summary of the Performance Evaluation

The performance of the Sloth implementation for the ARM Cortex-M3 was evaluated on multiple configurations of the system using three test applications with basic tasks, extended task, and an application with mixed types. The results were compared to Arctic Core, another implementation conforming to the AUTOSAR/OSEK specification. In all test cases in the three test applications measured and evaluated above, Sloth outperforms the Arctic Core implementation with a speed-up from 2.5 to 25.9.

Of course, the benefits of Sloth in an actual application depend on the executed task functions and the percentage of application code compared to the use of system services. As the

operating system usually is only used as a means to an end, as the applications running on them implement the actual functionality of a deployed system, it is important to have a low use of resources by the operating system combined with a high performance.

Nevertheless, the introduction of a single priority space for both tasks and ISRs by the SLOTH concept leads to additional advantages for real-time applications, as it allows to arbitrarily assign priorities among control flows without implying restrictions on the precedence of asynchronous over synchronous activation.

4.2 Comparison with the Reference Implementation on the TriCore Platform

The previous section evaluated the performance of the SLOTH implementation on the ARM Cortex-M3 platform, which was developed in the scope of this thesis. This new implementation founded on the design of the original SLOTH implementation that was written for the Infineon TriCore. This section compares the implementations for the two platforms and work out the reason for the differences.

4.2.1 The Infineon TriCore

The TriCore is a 32-bit platform, whose name derives from unifying three components: a real-time microcontroller, a digital signal processor, and a superscalar RISC architecture. The instruction set of the TriCore is usually encoded using 32 bit word length. The platform uses different registers for addresses and data, providing 16 registers with a 32-bit width for each type. This makes a total of 32 general purpose registers, which can be used freely with the exception of some reserved registers for stack pointer, return address and four global address registers. [5, 15]

Interrupt Subsystem of the Infineon TriCore

Of course, the Infineon TriCore includes an interrupt controller that fulfills the requirements defined in Section 1.4 [5, p. 1-8], which allows the implementation of the SLOTH concept. Different priorities can be assigned to each interrupt source, which can be triggered by either a connected hardware peripheral or by software. Additional interrupt sources are available for access from software only, which are not connected to any hardware device. The interrupt sources are organized in *service request nodes* (SRNs), which encapsulate the properties of the interrupts like priority, interrupt masking, and request state. An *interrupt control unit* (ICU) is responsible for determining the highest pending interrupt among all SRNs, which are connected to the ICU over a special bus exchange priority information. This *arbitration* takes a defined number of system bus cycles in parallel to normal CPU execution. The amount of cycles required for this calculation depends on the number of SRNs involved and their respective range of competing priorities. Thus, this arbitration takes fewer cycles with less ISRs configured.

The prioritization in the interrupt subsystem allows nesting of interrupts. A service request can interrupt the handling of another interrupt if the requested priority is higher than the currently handled interrupt. The precedence of an interrupt is always determined by the ICU, which signals this interrupt number to the CPU. Interruption of the current control flow executing on the CPU will only be carried out if the *current CPU priority number* (CCPN) is less than the priority of the requested interrupt. At exception entry, the context is stored in a fixed-size *context save area* (CSA), which is organized as a linked list of previously stored contexts. The interrupt handler is accessed using a vector table that contains the first few instructions of the ISR.

Summary

The Infineon TriCore platform fulfills the requirements for a SLOTH implementation and as a RISC architecture provides a similar set of features on the system level compared to the ARM Cortex-M3. However, the interrupt subsystem follows a different model by using a separate arbitration unit, which has implications on the implementation of task management and synchronization using resources.

4.2.2 Similarities and Differences in the Implementation of the System Services

Although the implementations for both platforms follow the same central idea of using the hardware for scheduling, the implementation varies due to the mentioned differences.

Basic Task Management

The basic task management on the TriCore is very simple as task activation boils down to triggering the respective interrupt source, just as it is on the ARM Cortex-M3. As already mentioned for the ARM Cortex-M3 in Section 3.2, synchronous task activation requires special attention.

As explained in the previous section, the interrupt subsystem on the TriCore requires arbitration cycles to determine the highest pending interrupt, which takes place in parallel to normal CPU execution. This is different on the ARM Cortex-M3, where pending an interrupt will be synchronized automatically with the data access to ISPR as one of the NVIC memory-mapped registers. On the TriCore, synchronous task activation will not be propagated to the CPU immediately due to the additional cycles required for arbitration and, thus, requires manual synchronization to avoid executing the instructions following the call to the `ActivateTask()` system service. This synchronization is achieved by executing NOP instructions while the arbitration runs in the ICU. The number of these NOPs is calculated statically during system generation to account for the worst-case latency caused by arbitration, depending on the number of arbitration rounds and the number of configured tasks and ISRs. Also, to define a synchronous point of preemption, before triggering the IRQ and the following NOP instructions interrupts need to be disabled. Enabling the interrupts afterwards lets the CPU handle the pending interrupt.

Such a synchronization mechanism is implemented for task chaining as well, where on the TriCore, interrupts are disabled to terminate the task before dispatching the chained task using the same NOP delay waiting for the arbitration. On return from the interrupt handler, interrupts are implicitly enabled allowing the chained task to be executed. On the Cortex-M3, this is solved similarly by using `FAULTMASK`, which raises the current execution priority over all configurable priorities. This is the only priority boosting method which is reset implicitly at a return from interrupt.

Extended Task Management

The extended task management uses a prologue on both systems to determine if the task has run before and has a context available for resume after waiting for an event. Additionally, stack switches happen at this point as each extended task has its own stack. In order to block an extended task in `WaitEvent()`, the implementation on the Cortex-M3 disables the interrupt source and then executes a return from interrupt. For this, the implementation on the TriCore achieves the same by disabling the interrupt source and then lowering the priority to zero, which yields the CPU and allows pending interrupts to be handled while the blocked task is not considered by

the ICU anymore. Again, synchronization using both NOPs and masking interrupts is applied to define the point of preemption. The implementation on the TriCore will never return to the previous task at the point it was preempted, instead, it relies on the prologue to restore the context of the task. On both systems, task unblocking enables the interrupt source again, which causes the interrupt controller to dispatch interrupt handlers and the corresponding blocked tasks based on their priorities in the current system state.

Basic tasks also need a prologue on both systems when used in conjunction with extended tasks. This prologue performs a stack switch when the basic task preempts an extended task. Additionally, the basic task needs to switch the stack when terminating with a return back to the preempted extended task. This requires checks whether a stack switch is necessary or not. Besides the different handling of terminating a task, this implementation detail is the same on both systems.

Resources Management

Resources are used to synchronize critical sections by raising the priority according to the OSEK priority ceiling protocol as described in Section 3.4. For raising the priority to the ceiling, the Sloth implementation on the Cortex-M3 uses `BASEPRI`, which allows to specify a minimum required priority for preemption, whereas the TriCore implementation modifies the current CPU priority level to the ceiling priority of the resource.

If an extended task on the TriCore is preempted, that task would be entered again using the task prologue, which sets the CPU priority level to that of the task and then restores the context saved at preemption. However, if a task had acquired a resource at the point of preemption, it is necessary to restore the raised priority as well. Therefore, the resource has its own IRQ assigned that is identified by the ceiling priority of the resource. This is calculated at system generation and is higher than the priority of all tasks accessing this resource. The generated prologue for this interrupt source of the resource determines which task had taken the resource at the time of preemption as recorded by the `GetResource()` system service. Leaving the CPU priority unmodified—as it already is at the resource priority—the resource prologue restores the context of the preempted task.

The implementation on the Cortex-M3 does not require any additional IRQ for resources, as the use of `BASEPRI` only defines the minimum required priority for preemption. The priority required for preemption on the Cortex-M3 is always the highest priority of all active exceptions and the `BASEPRI` value. This current execution priority is only compared to check if a pending interrupt can preempt the currently running control flow. Thus, interrupts with higher priorities can preempt a task with a resource acquired as usual as the `BASEPRI` will remain unchanged. The main difference is that, unlike on the TriCore, the prologue does not set the current priority level, which is derived from the active handlers instead.

Summary

The different hardware results in different implementations for the Infineon TriCore and the ARM Cortex-M3, especially in the task management. While the Sloth concept can be implemented on any hardware platform that fulfills the requirements as listed in Section 1.4, the actual implementations will be hardware-dependent as they rely on the hardware mechanisms provided to perform the scheduling. However, the implemented kernel is quite small and has an abstraction layer between the general OSEK functionality and the part relying on the hardware, making porting to other platforms easy.

Test Case	Task Type Transition	Stack Switch	SLOTH on TriCore
T1 ActivateTask	Basic → Basic	no	79
T2 ActivateTask	Basic → Extended	yes	116
T3 ActivateTask	Extended → Basic	yes	*n/a*
T4 WaitEvent	Extended → Basic	yes	168
T5 SetEvent	Basic → Extended	yes	118
T6 TerminateTask	Basic → Basic	no	29
T7 TerminateTask	Extended → Basic	yes	86
T8 TerminateTask	Extended → Extended	yes (dispatch new task)	94
T9 ChainTask	Basic → Basic	no (dispatch same task)	98

Table 4.4: Performance evaluation on the TriCore platform of task switching and blocking in a mixed task system with both basic and extended tasks, which requires stack switches for transitions involving extended tasks. The values specify the measured execution time in number of clock cycles. Numbers are taken from [3, Table IV].

4.2.3 Evaluation of the Test Cases

The SLOTH implementations for the Infineon TriCore and ARM Cortex-M3 have differences in their performance of the system services, but as they are implemented for distinct architectures, the measurements cannot be compared directly. Thus, this section compares the performance of system services between the systems itself to see where the different implementation influences the execution time. The measurements of the TriCore system are shown in Table 4.4, as taken from [3]. Test case T3 is not available for this system, as it was added to the test application for the measurements on the Cortex-M3 only.

There is almost no difference in additional execution time required for extended task activation compared to basic task activation between the two systems. On the TriCore, test cases T1 and T2 take 79 and 116 cycles, whereas the measurements on the Cortex-M3 for C1 and C2 took 42 and 65 cycles. This is a factor of 1.47 for the test cases T2 and T1, while on the Cortex-M3, the factor between C2 and C1 is 1.55. That means that the additional code for conditional checks and stack switches in the extended task prologue leads to similar results in relative execution timing for both platforms.

Examining the termination of tasks, a difference between `TerminateTask()` for basic and extended tasks can be observed in T6 and T7 in the implementation on the TriCore. The reason for this is the additional synchronization using NOPs required when terminating an extended task. In the TriCore implementation, extended tasks are not terminated by returning from the interrupt handler; instead, the priority of the interrupt is lowered to zero in the same way it is used for blocking a task. This yields the CPU and allows other interrupt handlers of higher priority to preempt this handler, but as that causes a new arbitration to find the interrupt of highest priority, which runs in parallel to CPU execution synchronized, it is required to synchronize this using NOP instructions. This results in a factor of 3.0 for termination of an extended task in T7 compared to termination of a basic task in T6. On the Cortex-M3, the termination of basic and extended tasks takes the exact same amount of cycles in C6 and C7 as they use the same code path as explained above in Section 4.1.

Another outstanding result is the test case for task blocking in C4 and T4, which is—on both systems—the slowest task transition of the test application. This is clearly caused by the fact

that the run-to-completion model of interrupt handlers does not fit blocking tasks. Hence, the implementation of `WaitEvent()` is the most extensive system service, which is reflected in its execution time.

Summary

It is difficult to compare the test cases between distinct architectures as the instruction set, addressing modes, pipelining, and other relevant properties of the microcontrollers influence the required execution cycles. However, the relative comparison of the test cases have shown that on both platforms, the task prologue of extended tasks requires the same factor of additional cycles when compared to basic task management. Also, the extended task termination by changing priority with manual synchronization of the interrupt arbitration on the TriCore leads to more additional cycles than on the Cortex-M3, where basic and extended task termination take the same amount of cycles.

4.2.4 Summary of the Comparison with the Reference Implementation on the TriCore Platform

The Sloth implementations on the Infineon TriCore and the ARM Cortex-M3 have similar runtime characteristics in most aspects of the task management, although the implementations have differences as they match the provided hardware functionality. However, both systems follow the same central design idea of using the hardware for scheduling purposes by making use of the interrupt controller. This is reflected in the evaluation of the test cases, where the relative comparison shows that the test scenarios take similar amounts of cycles.

4.3 Limitations

The port to the ARM Cortex-M3 platform is currently bound to the Atmel SAM3U microcontroller used in this thesis due to the use of the board-specific timer counters. Also, the startup code and initialization routines target this hardware configuration only. However, the Sloth implementation would be portable to any microcontroller using the ARM Cortex-M3 with minimal effort.

Also, the implemented Sloth system only supports the conformance classes BCC1 and ECC1 of the OSEK specification, as shown in Figure 1.1 on page 2. Of the additionally required features for the conformance classes BCC2 and ECC2, support for synchronization using resources and multiple activations of tasks have already been implemented. The missing feature is support for multiple tasks per priority. This is a limitation that cannot be solved easily, as the OSEK specification states that tasks of the same priority level have to be started in their order of activation. Thus, this requirement cannot be fulfilled by an interrupt subsystem without the use of a software scheduler determining which task will be dispatched next.

4.4 Summary

The performance evaluation confirms the Sloth concept, as it shows the advantages of utilizing hardware for scheduling. In comparison with another OSEK-conforming operating system, the Sloth implementation for the ARM Cortex-M3 outperformed the competitor with a software-based scheduler by a factor ranging from 2.5 to 25.9. The comparison with Sloth on the Infineon

TriCore shows that the different hardware influences the implementation details of the system services. However, the performance of the individual system services for task management are similar in the relative comparison.

Chapter 5

Conclusion

For this thesis, the Sloth concept of using interrupt controlling hardware for thread scheduling and dispatching purposes was examined on the ARM Cortex-M3 hardware platform. The goal of the Sloth concept is to omit a scheduler in software and use the interrupt subsystem instead by using interrupt handlers as a universal abstraction for control flows. The presented features of the Cortex-M3 platform were used to design and implement an operating system kernel based on the Sloth concept.

The implemented system was evaluated in different configurations and results were compared with another implementation using a software-based scheduler. With a speedup ranging from 2.5 to 25.9, the results show the positive effects of the Sloth concept onto the non-functional properties of the operating system.

The similarities and differences to the original Sloth reference implementation on the TriCore platform show that any Sloth implementation for a specific platform will be hardware-dependent in large parts, as each architecture has their own interfaces and peculiarities. However, the Sloth concept can be implemented on any platform that fulfills the basic requirements on the interrupt controller: different priority levels for each interrupt, support for triggering interrupts in software, and support for raising the current execution priority as a synchronization mechanism.

By modeling all control flows as interrupts in Sloth, the distinction between threads and interrupts is gone. Therefore, they can share the same priority space managed by the hardware, which avoids the problem of rate-monotonic priority inversion and allows assigning priorities arbitrarily between threads and interrupts.

The Sloth concept as described in this thesis could be implemented on any platform fulfilling the requirements on the hardware. In future work, Sloth needs to follow the ongoing trend of multi-core and multi-processor systems, which is spreading to the world of embedded systems at the moment. With multiple processor units, and therefore more complex scheduling decisions to make, a multi-core system would benefit from support by the hardware to improve the performance. Extending the Sloth concept for use in multi-processor systems will be an interesting challenge.

Bibliography

[1] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS 2006)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[2] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 204–213, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[3] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy Sloth: Threads as interrupts as threads. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011), to appear*, pages 67–77, Los Alamitos, CA, USA, December 2011. IEEE Computer Society.

[4] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`.

[5] Infineon Technologies AG. *TriCore 1 User's Manual, V1.3.8, Volume 1: Core Architecture*, January 2008.

[6] ARM. *Cortex-M3 Technical Reference Manual*, February 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337h/index.html`.

[7] ARM introduces the Cortex-M3 processor to deliver high performance in low-cost applications. `http://arm.com/about/newsroom/6750.php`, Oct 2004.

[8] ARM. *ARMv7-M Architecture Reference Manual*, November 2010.

[9] Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009.

[10] Atmel. *SAM3U Series*, March 2011. `http://www.atmel.com/dyn/resources/prod_documents/doc6430.pdf`.

[11] ARM. *Procedure Call Standard for the ARM Architecture*, October 2009. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/index.html`.

[12] Free Software Foundation, Inc. GCC 4.5 release series. Changes, new features, and fixes. `http://gcc.gnu.org/gcc-4.5/changes.html`, 2011.

[13] Free Software Foundation, Inc. Using the GNU compiler collection (GCC) – section 6.41: Assembler instructions with C expression operands. `http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`, 2011.

[14] ArcCore AB. Arctic Core - the open source AUTOSAR embedded platform. `http://arccore.com`.

[15] Infineon Technologies AG. *TriCore 1 User's Manual, V1.3.8, Volume 2: Instruction Set*, January 2008.