SLOTH:

The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems

Tugend und Laster der Latenzverbergung
in hardwarezentrischen Betriebssystemen

Der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg
zur Erlangung des Doktorgrades

Doktor-Ingenieur

vorgelegt von

Wanja Hofer

aus Ludwigshafen am Rhein

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	06.06.2014
Vorsitzende des Promotionsorgans:	Prof. Dr.-Ing. Marion Merklein
Gutachter:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr.-Ing. Jörg Nolte

Abstract

Software for embedded systems needs to be tailored to the application requirements to provide for the lowest cost overhead possible; this is especially important for embedded *operating systems*, which do not provide a business value of their own. State-of-the-art embedded operating systems *are* tailored to the requirements of the application above, but they abstract from the hardware platform below, missing out on advantageous hardware peculiarities to optimize the non-functional properties of the system. Additionally, they provide the application programmer with a multitude of control flow types, which leads to several severe restrictions and problems for the application development and in the real-time execution of the system—for instance, high-priority tasks can be interrupted by low-priority interrupt service routines at any time.

The SLOTH operating system design for event-triggered and time-triggered embedded real-time systems as presented in this thesis is unique in that it employs a hardware-centric approach for its scheduling, dispatching, and timing services and makes use of hardware particularities—with the purpose of optimizing the non-functional properties of the operating system and the application. In its implementation, SLOTH assigns each task an interrupt source with an appropriately configured priority and maps software activations of that task to setting the request bit of the interrupt source. This way, SLOTH has the hardware interrupt subsystem make the scheduling decision with the corresponding dispatch of the interrupt handler, which directly executes the application task function. Time-triggered dispatch tables are encapsulated in hardware timer cell arrays that SLOTH pre-configures at initialization time with the corresponding timing parameters, making the hardware timer subsystem execute the dispatcher rounds autonomously at run time.

The evaluation of the SLOTH operating system implementation shows that its light-weight design mitigates or even eliminates the safety-related problems and restrictions in the real-time execution of the application by providing a unified control flow abstraction in a unified priority space, eliminating the artificial distinction between tasks and ISRs. Additionally, its exhibited non-functional properties show unprecedented levels of efficiency by hiding latencies in the hardware subsystems: A feature-complete SLOTH operating system can be tailored to less than 200 lines of source code; it compiles to less than 500 bytes of code memory and as few as 8 bytes of data memory; and it can schedule and dispatch tasks in as few as 12 clock cycles with speed-up factors of up to 106 compared to commercial operating systems. Since SLOTH prototypically implements the automotive OSEK OS, OSEKtime OS, and AUTOSAR OS standards and runs on commodity off-the-shelf hardware, it is applicable to a wide range of embedded real-time systems.

Acknowledgments

Excelling at being lazy is hard work, and many people have encouraged and supported me in this objective.

First, I thank Wolfgang Schröder-Preikschat (Wosch) for his guidance in the research community in general and in several domains of operating systems research in particular. I am especially grateful for his personal support at the conference locations where I gave my publication talks.

Daniel Lohmann seamlessly shifted from being my diploma thesis supervisor to mentoring my PhD research. Pitching crazy/lazy ideas back and forth with him has proven to be invaluable for me when I was stuck.

Fabian Scheler helped me out when it came to real-time systems theory and its applicability. His sophisticated knowledge of the domain supported underlining my arguments both in publications and in my dissertation.

Jürgen Kleinöder always supported my development at the department starting when he pursuaded me to join as a teaching assistant back when I was still an undergrad student. After that, his involvement provided me the opportunity to do my study thesis at 3SOFT (now Elektrobit Automotive) and to organize my research stay at Stanford—showing me the extreme opposite of being lazy.

Peter Ulbrich, Timo Hoenig, and Julio Sincero have not only shown to be good colleagues but also to be good friends. Helping each other out in both research and personal matters made a huge difference for me.

Rainer Müller, Daniel Danner, and Markus Müller researched their diploma and Bachelor's theses under my guidance, helping to further develop the Sloth ideas. The former two have developed from being the disciples of Slothfulness to now being the preachers of Slothfulness at the department and in the research community.

Lots of colleagues and staff made my work at the university during the last years easier—and fun! I thank all of them: Johannes Behl, Niko Böhm, Daniel Christiani, Tobias Distler, Gabor Drescher, Christopher Eibel, Christoph Elsner, Christoph Erhardt, Meik Felser, Florian Franzmann, Michael Gernoth, Martin Hoffmann, Harald Junggunst, Rüdiger Kapitza, Tobias Klaus, Martin Mitzlaff, Judith Nopper, Benjamin Oechslein, Christian Preller, Matthias Schäfer, Jens Schedel, Horst Schirmeier, Benedikt Schultis, Guido Söldner, Olaf Spinczyk, Philippe Stellwag, Klaus Stengel, Isabella Stilkerich, Michael Stilkerich, Moritz Strübe, Reinhard Tartler, Christian Wawersich, Dirk Wischermann, Hao Wu, and Alexander Würstlein.

PHILIP LEVIS let me join his research group at Stanford; JOHN REGEHR from University of Utah also advised me during that time. All of the SING people made me feel welcome from the very beginning and made my stay worthwhile on an (inter) cultural and personal level: TAHIR AZIM, EWEN CHESLACK-POSTAVA, JUNG IL CHOI, MAYANK JAIN, MARIA KAZANDJIEVA, JUNG WOO LEE, BEHRAM MISTREE, and KANNAN SRINIVASAN.

My family—IRENA ZIMANJI-HOFER, HARALD HOFER, and ANNIE HOFER—and my family-in-law have always had my back and encouraged me to strike the balance between the ambitious side and the SLOTH side of the force.

VIOLA HOFER had the largest support package to carry, and she did it perfectly. I am usually rather eloquent when expressing myself, but I am failing to find the right words to pay tribute to her part in this. Thank you so much!

Aber es könnte doch auch sein, dass ein Monster erscheint und auf der Flöte spielt!

Contents

List of Figures

List of Tables

Motivation 1

> Sienna squinted at the text, reading it aloud. "Saligia?" Langdon nodded, feeling a chill to hear the word spoken aloud. "It's a Latin mnemonic invented by the Vatican in the Middle Ages to remind Christians of the Seven Deadly Sins. Saligia is an acronym for: superbia, avaritia, luxuria, invidia, gula, ira, and acedia." Sienna frowned. "Pride, greed, lust, envy, gluttony, wrath, and SLOTH."
>
> *From the book* Inferno *by Dan Brown, 2013*

In contrast to personal computers, which can be classified as general-purpose systems due to their versatility in possible uses, embedded systems are designed for special purposes only. These purposes are usually based on a tight integration of the system with the surrounding environment, which entails processing sensor input to produce output data for actuators such as an engine control. *Software* for embedded systems is different than software for personal computers in the sense that besides requirements on its *functionality*, requirements on the *non-functional* properties of the system are crucial in that domain. The software memory footprint, for instance, can decide whether a microcontroller derivative with smaller embedded memories can be used for the deployed system; the cents or fractions thereof saved in that process multiply by potentially millions of deployed devices. Additionally, real-time properties such as latencies and deadline guarantees are essential for most embedded systems, classifying them as *real-time systems*.

Operating systems help the application developer in writing complex software with high quality demands by offering services and abstractions for the decomposition of the application. An operating system also assists the application in ensuring real-time and other non-functional properties of the system and in using processor resources efficiently. The operating system is in a central position; it employs the microcontroller hardware platform below to implement services to be offered to the application (see Figure 1.1). In that process, non-functional properties also play an important role for the design and implementation of the embedded operating system: It needs to respect the non-functional *requirements*

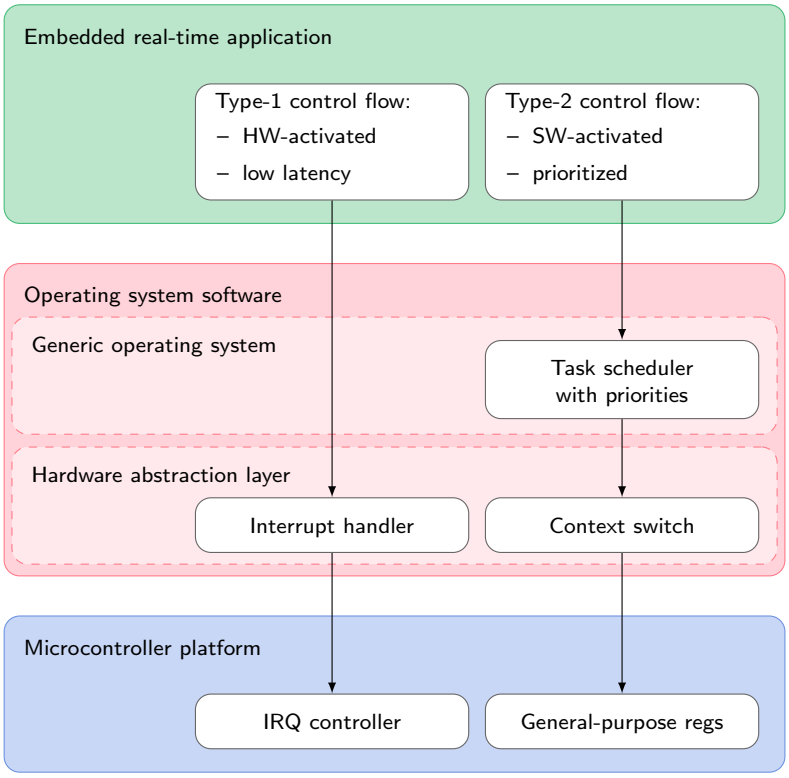


Figure 1.1: Central position of the operating system in an embedded real-time system. The operating system maps application control flow types to hardware facilities using an internal hardware abstraction layer.

by the application and it has to map the application functionality to match the hardware's non-functional *properties*.

The main responsibility of an embedded operating system is the management of control flows in the system on behalf of the embedded application. To be able to properly do that according to the application requirements, it therefore offers several control flow types with different non-functional properties for the application to implement its functionality in. For instance, functionality that needs to be performed with low latency upon the occurrence of a hardware event is usually encapsulated in a hardware-triggered interrupt service routine (ISR), whereas functionality that needs to be flexibly scheduled and prioritized is implemented in task control flows managed by the operating system scheduler.

When the embedded system developer has to choose from a variety of available operating systems and compare them against each other, his decision is usually based on the non-functional properties as advertised by the operating system—such as memory footprints, task activation and interrupt processing latencies, and support for deterministic real-time execution. This observation is supported by a recent study on embedded systems, where participants feel that

the operating system properties are crucial for the whole embedded system and where their objections and improvement wishes target non-functional properties of the operating system [UBM13]. Embedded operating systems' *functional properties*, on the other hand, are usually very similar in a given subdomain; the automotive industry, for instance, has even specified industry standards for operating systems in their domain, named OSEK OS [OSE05] and AUTOSAR OS [AUT13]. Those standards do not only prescribe the interfaces for operating system implementations, but also the corresponding system call *semantics* that the application can expect from an operating system implementing those standards. Hence, most embedded operating systems in the automotive sector can *only* be distinguished based on their non-functional properties. The non-functional properties of the operating system are subject to particular scrutiny, since operating systems do not provide a business value of their own and thus should not contribute negatively to the properties of the whole system.

Internally, many operating system implementations employ a hardware abstraction layer (HAL) [Tan07; Sta08], which is used by the services implementing the control flow types offered to the application and by other operating system services (see also Figure 1.1). Such an abstraction boundary is used to be able to port the complete operating system to another hardware platform by porting the HAL only and leaving the rest of the operating system implementation unaltered. To achieve this, the HAL interface offers abstractions that are generalized over all hardware platforms that the operating system runs on, hiding platform peculiarities from the rest of the operating system implementation.

In terms of non-functional properties, this state of the art in the design of embedded real-time operating systems causes two prevalent problems, which are discussed in a comprehensive problem analysis in Chapter 2:

1. The multitude of available control flow types to satisfy the application's non-functional requirements leads to real-time problems, restrictions, and inflexibilities for the embedded real-time application.

2. The internal hardware abstraction layer to increase operating system portability hides potential to optimize the operating system's non-functional properties, which are directly perceived by the embedded real-time application.

1.1 Real-Time and Optimization Problems in State-of-the-Art Embedded Operating Systems

The approach to offer a specific control flow type to satisfy distinct requirements on non-functional application properties bears several severe problems for the application development process as well as for the deployed embedded real-time application. Different control flow types are also scheduled according to different, independent priority spaces, which is an inherent challenge for real-time analysis

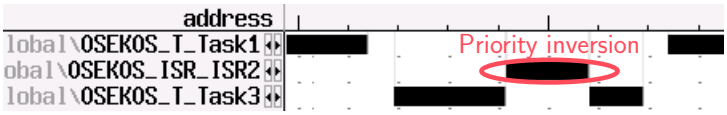


Figure 1.2: Occurrence of rate-monotonic priority inversion in a commercial embedded real-time operating system. The execution trace shows a medium-priority interrupt (with its handler `ISR2`) occurring during the execution of a high-priority task `Task3` in a commercial OSEK OS system. `Task3` is interrupted and `ISR2` is executed immediately—leading to rate-monotonic priority inversion.

and predictable real-time execution. For instance, in state-of-the-art embedded real-time operating systems, an ISR control flow is always able to interrupt and disturb the execution of a task control flow—although application requirements might demand the task have a higher *semantic* priority than the ISR.

Figure 1.2 shows the execution trace of a real-time application with a low-priority task `Task1`, preempted by a high-priority task `Task3`, and the occurrence of a medium-priority interrupt service routine `ISR2`. If the corresponding real-time application is executed on a state-of-the-art commercial embedded real-time operating system—which the trace was retrieved from—the high-priority task is interrupted for the ISR of lower priority, potentially making it miss its deadline and hampering real-time analysis before run time. This phenomenon has been termed *rate-monotonic priority inversion* [FMAN06a]—one type of the general phenomenon of priority inversion as exhibited in real-time systems (see also background in Section 2.2.1)—and is one of the gravest problems induced by the design of current embedded real-time operating systems. Due to the existence of rate-monotonic priority inversion, real-time application programmers are taught to keep ISRs short and to have most functionality execute in post-processing tasks. In general, the multitude of offered control flow types in current embedded operating systems leads to potential real-time problems, inflexibility, and unnecessary restrictions in the development and deployment of applications for those operating systems; the exact reasons and circumstances are analyzed in Section 2.2. Related work has also analyzed and recognized the implications of different priority spaces, particularly rate-monotonic priority inversion (see comprehensive discussion of related work in Section 7.3.1).

Since operating systems provide no business value of their own but do consume resources of the embedded system, current embedded operating systems are implemented as *configurable software* and offer configurable features to the application. This way, the operating system can be tailored *upward* to offer exactly the kind of functionality that the application needs. By excluding from compilation the operating system code and data that is not needed by the application, the operating system binary is optimized in the direction of several non-functional properties, including system call latencies and RAM and ROM memory footprint. *Downward*, however, the state of the art in embedded operating system

design employs HALs to accommodate different microcontroller platforms to facilitate the portability of the operating system itself. Thus, instead of tailoring the operating system to the hardware, it abstracts from particular beneficial features that the hardware offers and that could otherwise be used by the application or the operating system itself in order to optimize the non-functional properties of the whole embedded system—although some of the advantages of tailoring the operating system to the hardware have been recognized by related work (see discussion in Section 7.3.2). Additionally, the HAL abstracts from possible performance bottlenecks pertaining to the platform, which therefore can neither be respected by the design and implementation of the application nor by the one of the operating system itself, potentially yielding unfavorable performance properties on a subset of hardware platforms. That one-size-fits-all solution leads to missed potential for optimization since the set of microcontrollers in use by embedded projects is very diverse, still ranging from 8-bit to 32-bit and even 64-bit controllers (see results of a recent study on that topic in [UBM13]).

In his seminal paper "Hints for Computer System Design", Butler W. Lampson already stated "not to hide power" [Lam83] (see also more comprehensive discussion in Section 7.3.2):

> When a low level of abstraction allows something to be done quickly, higher levels should not bury this power inside something more general. The purpose of abstractions is to conceal *undesirable* properties; desirable ones should not be hidden.

State-of-the-art embedded operating systems, however, *do* hide the power of the underlying microcontroller platform for the sake of operating system portability at the expense of disadvantageous non-functional operating system properties. The exact influence and problems induced by current operating system HAL designs are analyzed and discussed in Section 2.3.

1.2 Goals of This Thesis

The overall goal of this thesis is therefore to develop an embedded real-time operating system design that offers flexible and unrestricted control flows for the application running on top and that is tailored to the properties of the hardware platform located below—in order to improve the non-functional properties of the whole embedded system. This entails

1. facilitating the choice the application developer has to make by unifying control flow types to fewer and more universal abstractions—without sacrificing the possibility to optimize non-functional properties;

2. unifying previously separate priority spaces to seamlessly reflect the application requirements on control flow priority distribution—thereby preventing situations of rate-monotonic priority inversion *by design*;

3. and re-evaluating the abstraction boundary of the hardware abstraction layer that is internal to the embedded operating system in order to expose advantageous non-functional properties of the hardware platform below to the applications above instead of hiding hardware features.

The resulting hardware-centric operating system shall run on commodity off-the-shelf microcontroller platforms and shall target embedded hard real-time systems with static control flow priorities, which enables real-time schedules such as the rate-monotonic [LL73] or the deadline-monotonic algorithm [Liu00]. Furthermore, the implemented operating system shall exhibit favorable non-functional properties that are important in the embedded real-time domain: low event latencies, fast and deterministic system calls, and low memory footprint in RAM and ROM.

1.3 Proposed Approach

The approach to reach those goals is reflected in the design of the SLOTH operating system[1], which is detailed in the rest of this thesis, and entails the following principles, which are detailed in Chapter 3:

Two-dimensional operating system tailoring
In addition to tailoring the operating system implementation to the application requirements, the SLOTH operating system is also tailored to the features that the hardware platform offers by using generative techniques.

Hardware-centric operating system design
Latencies induced by the operating system are hidden in SLOTH by an operating system design that respects hardware peculiarities to out-source operating system functionality to platform components as much as possible.

Hardware-centric control flow design
Control flow abstractions and corresponding application synchronization mechanisms that are offered by the SLOTH operating system in its interface and the corresponding schedulers and dispatchers are implemented using hardware abstractions and platform facilities as far as possible.

Hardware-centric timing design
Timing requirements imposed by time-triggered applications are directly mapped to hardware timer configurations by the SLOTH operating system as far as possible.

[1] The name honors both the fourth deadly sin of laziness and the lazy animal breed.

Flexible hardware abstraction
SLOTH internally provides adaptable hardware abstraction by featuring a higher-level hardware abstraction interface together with a flexible hardware abstraction layer that the SLOTH framework adapts both to the hardware and to the application at compile time.

1.4 Structure and Contributions of This Thesis

The findings and contributions of this thesis are structured as follows.

Chapter 2 comprehensively analyzes the real-time and optimization problems that can be perceived in state-of-the-art embedded real-time operating systems. It finds that those problems are rooted in the multitude of available control flow types and in the way that traditional operating systems employ hardware abstraction layers.

Chapter 3 presents the SLOTH approach, whose novel design principles make the hardware platform a first-class element to consider in the operating system design. The approach is based on tailoring the operating system in two dimensions, on designing the operating system in a hardware-centric way, and on providing flexible hardware abstraction *inside* the operating system design.

Chapter 4 develops the novel hardware-centric SLOTH design, which relies on hardware subsystems for the purposes of the operating system as much as possible. The design description presents the event-triggered SLOTH and SLEEPY SLOTH operating systems, which rely on the interrupt subsystem, and the time-triggered SLOTH ON TIME operating system, which additionally relies on the timer subsystem of the underlying hardware.

Chapter 5 shows how the SLOTH design is implemented using generative techniques and how the operating system is tailored both upward to the application as well as downward to the hardware platform. It also shows how its implementation structure enables the operating system to be highly hardware-optimized and to be kept small and fast using implementation examples from the SLOTH reference platform, the Infineon TriCore.

Chapter 6 presents the results of the evaluation of the hardware-centric SLOTH design and implementation. The results include unprecedented non-functional properties in operating system code and data size (as few as 96 bytes of code and 8 bytes of data for a minimal application), performance and latency benchmarks (as few as 13 clock cycles for a task switch), lines of source code in the operating system (as few as 200 lines), and its susceptibility to rate-monotonic priority inversion (prevented by design).

Chapter 7 discusses the results, stating advantages of the SLOTH approach as well as its applicability and limitations and how to overcome them. Additionally, it comprehensively presents related approaches from other pieces of scientific work, most of which have been briefly discussed in the previous chapters, and it analyzes the achievement of the SLOTH design goals and how the SLOTH approach addresses the initially stated problems.

Chapter 8 concludes the thesis by summarizing its results, findings, and contributions, and by providing an outlook on on-going and future work and on the broader applicability and evolution of the SLOTH approach.

1.5 Context and Terminology

The SLOTH approach and its design principles presented in this thesis are applicable to a range of operating systems of different domains. The SLOTH prototype and its description in the thesis text, however, focus on the domain of *embedded real-time operating systems*; the text provides an outlook on the implications of the SLOTH approach for other domains where applicable. Note that this thesis is concerned with the design and implementation of real-time *operating systems*; real-time *applications* and their designs, their decompositions into co-operating application control flows, and their worst-case execution time calculations, for instance, are not in the scope of this thesis.

Since real-time theory and real-time operating system implementations use a variety of terms with differing meanings, I consistently use the terminology provided by the existing automotive industry standards OSEK OS [OSE05], OSEK-time OS [OSE01], and AUTOSAR OS [AUT13] as a continuous example throughout this thesis. More importantly, however, the considerations in this thesis use the *requirements* on an embedded real-time operating system as specified by those external standards as an example—in order to be able to discuss the implications of the developed SLOTH approach and its abstractions in a way that is *unbiased* by the approach itself. Since real-time theory and implementations use a wide variety of abstractions and execution models, this restriction makes a discussion of the applicability of the SLOTH approach viable and allows for a fair evaluation of the SLOTH implementation compared to other implementations of those standards with similar functionality. Again, the text provides an outlook on the implications of the SLOTH approach on real-time abstractions beyond those automotive standards where applicable.

This section briefly introduces those parts of the terminology and semantics used by the SLOTH real-time operating system that are necessary to follow the rest of this thesis; the terms are explained in more detail when required. The target class of embedded real-time operating systems is configured completely statically, with control flows that are statically created at compile time with static priorities, and with a static configuration of available features. Operating systems of the embedded real-time class usually feature system calls that are implemented

as simple function calls for efficiency reasons (i.e., not as trap exceptions as in desktop operating systems).

As an example, Figure 1.3 shows a diagram of the features and abstractions that are potentially included in an event-triggered OSEK OS real-time operating system, used by the discussions in this thesis and by the SLOTH implementation. Application code is encapsulated in *basic tasks*[2], which run to completion and preempt each other according to their statically defined priorities if the system is configured to be fully preemptive. *Category-2 ISRs* are activated asynchronously by hardware, are allowed to access the operating system via system calls, and therefore need to be synchronized with the operating system in order not to corrupt system state. *Category-1 ISRs* are not allowed to issue system services[3] and therefore potentially exhibit a lower worst-case latency. Application tasks can synchronize for mutual exclusion in a critical section via *resources*, which adhere to a stack-based priority ceiling protocol, raising the task priority to the statically determined ceiling priority in the critical section. If a task is statically configured by the application developer to be an *extended task*, it can block while waiting for an *event* for signaling purposes; events are stored in a task-local event mask by the operating system. Extended tasks correspond to what some real-time system literature calls *complex tasks* and to what other operating systems call kernel level *threads*—blocking control flows whose state needs to be saved in a separate memory section. Most implementations therefore assign extended tasks stacks of their own. In contrast to preemption, blocking disrupts the stack-based task execution pattern by temporarily executing lower-priority tasks until the high-priority task is unblocked. Time-based task activation or signaling is performed via the *alarm* abstraction in OSEK OS systems or via the *schedule table* abstraction in AUTOSAR OS systems. Depending on which minimum feature sets have to be supported by the operating system, OSEK OS specifies the conformance classes BCC1 and BCC2 for systems with only basic tasks, and ECC1 and ECC2 for systems that support extended tasks (see also Figure 1.3).

Time-triggered systems[4] schedule periodic tasks by computing a static schedule offline for one hyperperiod of the tasks and by dispatching those tasks according to the pre-computed schedule at run time. The SLOTH ON TIME prototype presented in this thesis adheres to that model and calls the hyperperiod its *dispatcher round* (as does the OSEKtime specification). Additionally, it accommodates static deadlines for its tasks, which the scheduler reviews at run time for violations.

[2]Real-time system literature features a variety of notions for the terms job, task, and thread. This thesis and the SLOTH operating system use the term *task* as the basic unit of computation that can be activated synchronously by software and can thus be scheduled and dispatched by the operating system scheduler.

[3]The operating system can check for violations of such restrictions by using static-analysis tools and static code assertions where possible, and by resorting to dynamic checks otherwise. Those checks can be configured to be omitted in production system, which might demand for optimization of memory footprint and reaction latencies. This thesis uses the expression “not allowed” to describe such restrictions.

[4]Literature also calls them clock-driven or timer-driven systems.

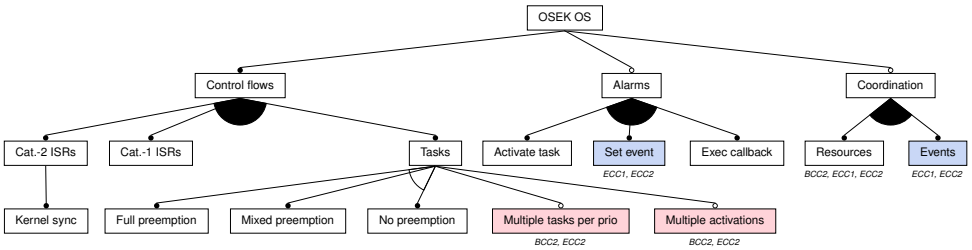


Figure 1.3: Feature diagram of an OSEK OS operating system. Feature types include mandatory features (filled circle), optional features (hollow circle), minimum-one feature sets (filled arc), and exactly-one feature sets (hollow arc). If a particular feature is mandatory only in conformance classes other than the basic BCC1, this information is given below that feature.

Time-triggered models that aim to further improve determinism employ strictly periodic *frame boundaries* for the tasks to be dispatched at; at run time, the corresponding *cyclic executive* monitors whether all tasks or slices thereof scheduled in the current frame have been released and whether it detects an overrun of the previous frame. Further enhancements include queues for aperiodic tasks that are executed in the background, acceptance tests for sporadic tasks with hard deadlines, and slack stealing techniques to improve the average response time of those aperiodic and sporadic tasks.

Figure 1.4 presents an overview of the introduced standards and their relationships. The rest of this thesis presents the SLOTH operating system prototype, which mainly targets run-to-completion task systems, the SLEEPY SLOTH operating system, which mainly targets systems with blocking tasks, and the SLOTH ON TIME operating system, which mainly targets time-triggered systems and systems with time-based task activations. The term SLOTH is also used as a super term for all of those operating systems and for the overall approach where appropriate in the corresponding context.

1.6 Related Publications

Parts of this thesis have been published in the following conferences:

[HLSSP09a] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. “Sloth: Let the Hardware Do the Work”. In: *Proceedings of the Work-in-Progress Session of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. (Big Sky, MT, USA, Oct. 11–14, 2009). ACM Press, Oct. 2009.

[HLSSP09b] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. “Sloth: Threads as Interrupts”. In: *Proceedings of the 30th IEEE International Symposium on Real-*

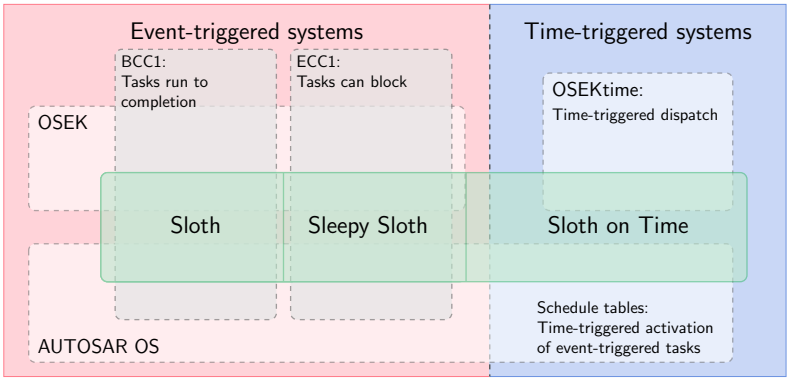


Figure 1.4: Overview of types of automotive real-time systems and the corresponding main domains of SLOTH operating systems. Systems with run-to-completion tasks are covered by SLOTH, systems with blocking tasks are covered by SLEEPY SLOTH, and time-triggered systems and systems with time-based task activations are covered by SLOTH ON TIME.

Time Systems (RTSS '09). (Washington, D.C., USA, Dec. 1–4, 2009). IEEE Computer Society Press, Dec. 2009, pp. 204–213. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.

[HLSP11] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads". In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS '11)*. (Vienna, Austria, Nov. 29–Dec. 2, 2011). IEEE Computer Society Press, Dec. 2011, pp. 67–77. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.14.

[HDMSSPL12] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS". In: *Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS '12)*. (San Juan, Puerto Rico, Dec. 4–7, 2012). IEEE Computer Society Press, Dec. 2012, pp. 237–247. ISBN: 978-0-7695-4869-2. DOI: 10.1109/RTSS.2012.75.

In [HLSSP09a; HLSSP09b; HLSP11], I was the leading author of the publication and the main contributor to the design and implementation of the system. In [HDMSSPL12], I was the leading author of the publication and the main contributor to the design of the system.

12 ■

Problem Analysis 2

Mac is giving confession.

Mac: [...] Dee interrupted me with some lazy plan to get out of working. Lazy is a sin, right?
Priest: You mean SLOTH.
Mac: Yeah, yeah, yeah, yeah. Dee is guilty of SLOTH. And she is pro abortion.

From the TV series It's Always Sunny in Philadelphia*, episode S07E10, 2011*

As presented in the previous chapter, current operating systems feature a multitude of different control flow types for the application to use. This chapter presents an analysis of the necessity for those types (see Section 2.1), together with resulting problems, restrictions, and inflexibilities for the application developer (see Section 2.2)—exemplified by the domain of embedded real-time operating systems, which is the focus of the SLOTH prototype and of this thesis. Additionally, the reasons for the design of state-of-the-art hardware abstraction layers in embedded real-time operating systems are analyzed, together with a presentation of the problems introduced by this approach (Section 2.3).

2.1 Control Flow Types in Embedded Real-Time Operating Systems

Since current embedded real-time operating systems offer different control flow types to be able to reflect different non-functional requirements, software engineers who want to develop an embedded real-time application need to choose between those types to decompose their software project. Available control flow types typically include tasks with particular preemption and blocking properties, different kinds of ISRs, and callback functions, among others, and they differ in the *non-functional properties* that they exhibit. Control flow types are subject to trade-off decisions: Some control flow types have lower execution overheads and activation latencies but are not allowed to access operating system state via system calls, whereas others are allowed to use operating system services but

Control flow type	Activation	Latency	Preemptability	Prioritization	Priority order
Category-1 ISR	by HW	very low	dep. on HW	dep. on HW	dep. on HW
Category-2 ISR	by HW	low	dep. on HW	dep. on HW	dep. on HW
Preempt. basic task	by SW	high	yes	yes	arbitrary
Non-preempt. basic task	by SW	very high	no	no	none
Extended task	by SW	very high	yes	yes	arbitrary
Time-triggered task	by HW timer	low	yes	no	none
Callback function	by HW	low	no	no	none

Control flow type	OS access	Scheduling and dispatching	Synchronization mechanism	Execution flexibility	Stack sharing
Category-1 ISR	no	by HW	IRQ disabling	very low	yes
Category-2 ISR	yes	by HW and OS	OS lock, (res.)	low	yes
Preempt. basic task	yes	by OS	sched. lock, res.	high	yes
Non-preempt. basic task	yes	by OS	n/a	high	stack re-use
Extended task	yes	by OS	sched. lock, res.	very high	no
Time-triggered task	no	by HW and OS	n/a	low	yes
Callback function	no	by OS	n/a	very low	yes

Table 2.1: Typical control flow types offered by embedded real-time operating systems, and their properties. The list is exemplified by the OSEK OS [OSE05] and OSEKtime OS [OSE01] standards, and shows their functional and non-functional properties as perceivable by the application, including functionality provided by the hardware (HW), the software (SW), and the operating system (OS).

can only be activated in software, for instance. Since there is no one-size-fits-all abstraction in a typical embedded real-time operating system, the software developer needs to carefully assess his requirements on the designed application control flows and needs to decide which control flow type to use for each of them.

To give an example of typically available control flow types and properties, Table 2.1 shows an overview of the types offered by embedded real-time operating systems implementing the OSEK OS [OSE05] and OSEKtime OS standards [OSE01] specified by the automotive industry. Each of the control flow types has distinct characteristics in terms of deployment possibilities—for instance, *hardware-activated* control flows *must* be ISRs or callback functions—and non-functional properties—like execution latency, preemptability by higher-priority control flows, or the possibility for stack sharing. Thus, in embedded real-time applications developed for the automotive industry, the application engineer has to choose between seven control flow types, which differ in at least ten functional and non-functional properties as assessed and summarized in Table 2.1:

Activation

Some control flows can only be activated asynchronously by hardware pe-

riphery or the timer, whereas others are activated synchronously in software using a system call such as posting a semaphore, for instance.

Latency
Some control flows are executed right after the corresponding event has been triggered, whereas others are subject to higher software overhead (dispatch latency); additionally, some control flows can be delayed by critical operating system sections or by concurrently running control flows (scheduling latency).

Preemptability
Some control flows cannot be preempted by other control flows of the same type, whereas others *are* preemptable.

Prioritization
Some control flows can be prioritized in their execution compared to other control flows of the same type, whereas others cannot.

Priority order
Some control flows have fixed priority slot assignments (depending on the hardware platform), whereas others can be configured in their priorities.

Operating system access
Some control flows have access to operating system services via system calls, whereas others are not allowed to access the operating system—a property that the operating system can usually be configured to enforce statically or dynamically if needed by the application.

Scheduling and dispatching
Some control flows are scheduled and dispatched directly by the hardware platform, whereas others are subject to policies imposed by the operating system.

Synchronization mechanism
Some control flows synchronize with other control flows of the same type by disabling interrupt requests (IRQs) or via a resource abstraction, whereas others use the operating system lock or the scheduler lock provided by the operating system.

Execution flexibility
Some control flows have the flexibility to block in their execution while retaining their execution state, whereas others must run to completion once started.

Stack sharing
Some control flows allow for sharing their stack with other control flows to

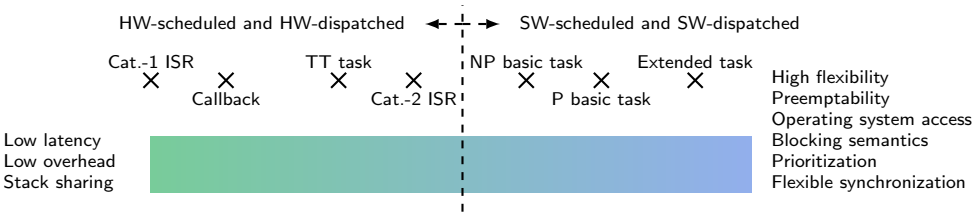


Figure 2.1: Simplified example spectrum of control flow types and their properties. As an example, the figure shows control flow types from the OSEK OS and OSEKtime OS specifications; hardware-oriented control flow types tend to exhibit lower overhead, whereas software-oriented control flow types are generally more flexible.

reach higher memory efficiency, whereas others need to execute on a stack of their own.[1]

Embedded real-time application developers therefore have a wide variety of control flow types to choose from in order to exploit the diverse control flow properties; additionally, some of those properties have an effect on other properties or even imply them. Thus, part of their engineering task is to write control flow code and to map each control flow to an appropriate available execution environment of a certain type. In general, the different control flow types trade efficiency for flexibility, besides certain other properties; Figure 2.1 shows a simplified overview of the spectrum of the analyzed control flow types and their properties. Control flow types whose management is more closely connected to the hardware are generally more efficient in their performance and latencies but less flexible compared to those that are activated and scheduled by the operating system software.

The depicted control flow types and properties reflect the OSEK standards as an example, which are themselves very concise standards; however, they still have to offer that many control flow types in order to fulfill the whole variety in the requirements spectrum. For instance, if the embedded real-time application wants to optimize its non-functional property of RAM usage, it can use basic tasks, which run to completion and can thus share their stacks among each other. If it wants to optimize stack usage even further, it can rely on non-preemptable basic tasks so that only the *maximum* stack used by all non-preemptable tasks has to be allocated in RAM. If the application needs to react on timer expiries with lower latencies than would be possible with scheduling and dispatching a task, it can use small and light-weight callback functions, which are directly executed with very low overhead.

[1] Note that blocking control flows do not necessarily need a stack of their own, although this is a common operating system design and also employed by the SLOTH operating system. With operating system abstractions like Mach continuations [DBRD91] or protothreads [DSV05; DSVA06], a blocking control flow can save its state to another dedicated memory area while it remains blocked.

The main problems caused by the variety of control flow types is actually caused by the variety of control flow *priority spaces*, which lead to control flows being scheduled and dispatched in their type class but not globally in a system; this fact has also been recognized by related work (see discussion in Section 7.3.1). In typical embedded real-time systems, for instance, there are four priority spaces: first-level interrupts (also called category-1 ISRs) are scheduled and prioritized by the interrupt controller, second-level interrupts (also called category-2 ISRs or bottom-half handlers) are scheduled non-preemptively by the operating system, tasks are scheduled preemptively by the operating system scheduler, and the application often has user level schedulers in order to execute light-weight events within an application task [RRWPL03]. Some hardware platforms increase the effect by offering an additional control flow type for "fast" interrupt handlers; many ARM microcontrollers, for instance, have so-called fast IRQs (FIQs), which are scheduled and dispatched with a higher priority and lower latency compared to regular IRQs [ARM01]. However, most real-time problems are due to the existence of *at least* two kinds of control flows and two priority spaces—typically, these are at least tasks and ISRs.

2.2 Real-Time Problems, Restrictions, and Inflexibilities

The presented multitude of available control flow types in state-of-the-art embedded real-time operating systems leads to a number of problems, restrictions, and inflexibilities for the developer of such embedded real-time systems. Since a majority of those problems relates to priority inversion in real-time systems and to the synchronization of the operating system and the application, those are first put into a general context before detailing the analyzed restrictions.

2.2.1 Background: Priority Inversion and System Synchronization

Priority inversion is a phenomenon that occurs when a lower-priority control flow is being executed although a high-priority control flow is ready to run. Situations of priority inversion are especially considered in the design of priority-based real-time systems, whose analyses are generally based on the assumption that the system adheres to the configured control flow priorities. In some situations, however, priority inversion cannot be avoided: For instance, if a high-priority task wants to access a resource that it shares with a low-priority task that is currently in the corresponding critical section, the high-priority task is kept from executing. Such a situation is called *bounded* priority inversion, since the priority inversion time is bounded by the amount of time it takes the low-priority task to leave its critical section. This type of priority inversion is influenced by the design of the application itself, its shared resources and its critical sections, and by the accesses to those critical sections by its tasks with the corresponding priorities.

Unbounded priority inversion occurs when medium-priority tasks preempt the low-priority task and therefore indirectly keep the high-priority task from running

longer as necessary [LR80], and for an amount of time that is unbounded in advance and is therefore difficult to be taken into account by real-time analyses. This type of priority inversion can be avoided by the system by using different protocols for maintaining critical sections, including priority inheritance protocols [SRL90] or priority ceiling protocols (one variant of which is also used by SLOTH; see Section 4.2.3), for instance. *Rate-monotonic* priority inversion, which was introduced in the motivation in Section 1.1, is triggered by the occurrence of low-priority interrupts during the execution of high-priority tasks. It is also unbounded in the sense that a high-priority task can be kept from executing by an arbitrary amount of time allocated to low-priority ISRs, again hampering real-time analyses. Thus, one of the main design goals of the SLOTH system, besides avoiding unbounded priority inversion, is to prevent rate-monotonic priority inversion.

A topic that is closely related to priority inversion is the way that the computing system is synchronized, which can be split into application synchronization and operating system synchronization. The application synchronizes its concurrent control flows, which usually include tasks and ISRs, and their accesses to shared resources and critical sections using mechanisms offered by the operating system in order to manage potential race conditions. These mechanisms can include monitors and mutual exclusion abstractions, which are either implemented by using blocking synchronization techniques or by using non-blocking synchronization constructs.

Blocking synchronization delays the execution of potentially conflicting control flows during a critical section—for instance by locking interrupts. This approach is simple and efficient since the enter and leave mechanism for a critical section can usually be implemented with few machine instructions. However, blocking synchronization increases the worst-case latencies for the involved control flows, since they are always kept from executing during a critical section, although they might not even enter the same critical section at that point.

Non-blocking synchronization does not explicitly delay conflicting control flows and mostly works by executing on a copy of the shared data and by atomically checking for interruption and committing the changes using a special hardware instruction such as compare-and-swap (CAS), which is an example for an atomic read–modify–write primitive. If the hardware platform does not offer CAS-like instructions, the implementation has to resort to locking interrupts, yielding the same disadvantages as with blocking synchronization to begin with, plus additional overhead and complexity for the non-blocking algorithm. Depending on its properties, a non-blocking synchronization construct is called lock-free if it guarantees progress on the system level, and it is called wait-free if it guarantees progress for each of its control flows [Her91]. The advantage of non-blocking synchronization is that it does not keep concurrent control flows from executing; however, a control flow interrupted in its critical section will have to roll back and re-execute it. Additionally, real-time systems as in the scope of this thesis need *wait-free* synchronization to be able to guarantee real-time pro-

perties, and such wait-free non-blocking synchronization algorithms can be hard to develop depending on the particular use case. In some cases, the overhead introduced by the wait-free synchronization protocol will surpass the length of the critical section itself by a large factor, mitigating its advantages compared to blocking synchronization.

Since it manages synchronous task control flows and concurrent ISR control flows, the operating system itself also needs to be synchronized to keep its internal data structures, such as its ready queue, in a consistent state when a control flow accesses them in a system call. As in the application synchronization, this can be reached by blocking means—delaying the execution of concurrent control flows—or by non-blocking means—developing an algorithm that performs an atomic commit at the end. For operating system synchronization, non-blocking synchronization has the advantage that it does not delay concurrent ISRs, but it introduces jitter to the execution of system calls, which might need to roll back and re-execute if they have been interrupted during their critical sections. Due to its design, SLOTH only has few data structures that it needs to protect (see also Section 5.5.5); it uses blocking mechanisms due to their simplicity, the unavailability of CAS-like instructions for the non-blocking commit operation on many microcontroller platforms, and due to the overhead introduced by wait-free synchronization as argued above. Since the focus of the SLOTH design is on resource-constrained embedded real-time systems, blocking synchronization better fits the corresponding requirements.

The rest of this thesis focuses on the operating system synchronization as well as on the priority inversion induced by the operating system and the abstractions it offers. Priority inversion effects caused by the *application design* are not in the scope of this thesis.

2.2.2 Problems Resulting from the Multitude of Control Flow Types

The problems induced by the availability of a multitude of control flow types can be grouped into problems related to priority inversion situations, system synchronization problems, restrictions in the development of the embedded real-time application, and inflexibilities in the evolution and modularization of the application. This section lists and briefly explains those problems and their roots; additionally, a short note states why the SLOTH design approach presented in the rest of this thesis can overcome a particular problem. The exact explanation as to *how* SLOTH overcomes those problems is detailed in the corresponding part of this thesis. A subset of those problems has also independently been identified by Kleiman and Eykholt [KE95], Regehr et al. [RRWPL03], and Leyva-del-Foyo et al. [FMAN06a; FMAN06b; FMAN12], amongst others. Work related to the problems due to multiple control flow types and proposed solutions are comprehensively discussed in Section 7.3.1.

1. **Priority inversion**

   a) **Direct rate-monotonic priority inversion**
   In embedded real-time applications featuring tasks and ISRs, high-priority software implemented in software-activated tasks can be interrupted and disturbed by low-priority software implemented in hardware-activated ISRs; this phenomenon is termed rate-monotonic priority inversion [FMAN06a]. This causes a loss in the system scheduling precision and jitter in the execution of high-priority tasks, which in the worst case might even miss their deadlines.

   In SLOTH, tasks and ISRs run in a single priority space—the interrupt priority space—preventing rate-monotonic priority inversion *by design*.

   b) **Indirect rate-monotonic priority inversion**
   Other occurrences of rate-monotonic priority inversion include situations where *operating system IRQs* interrupt high-priority application control flows *on behalf* of low-priority application control flows. Consider, for instance, the situation when the operating system timer IRQ—which usually has one of the highest priorities in the whole real-time system—interrupts a high-priority application task in order to activate a low-priority application task. The execution of the high-priority task is then delayed by operating system functionality executed on behalf of a low-priority task, yielding a situation of indirect rate-monotonic priority inversion. This situation cannot be avoided if the operating system uses a single system timer in its design, since it cannot know beforehand which priority the task has that a specific timer IRQ will activate.

   In SLOTH ON TIME, timer-based task activations are directly scheduled and dispatched according to the priority of the activated task.

   c) **Indiscriminate use of IRQs and ISRs**
   Since studies indicate that major problems in embedded real-time systems are caused by IRQs and ISRs being used by many developers of real-time applications in an inconsiderate manner, the effect of rate-monotonic priority inversion is amplified in real-world systems.
   Quoting David B. Stewart from his "Twenty-Five Most Common Mistakes with Real-Time Software Development" [Ste99]:

   > Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements. The reason is that interrupts preempt everything else, and are not scheduled. If they preempt a regularly scheduled event, undesired behavior may occur.

In SLOTH, the use of long-running ISRs is as legitimate as is the use of long-running tasks.

d) Priority inversion in mixed-criticality systems
If several subsystems or applications with mixed criticality execute on one microcontroller, they cannot be effectively shielded from one another due to the intertwined task and ISR priority spaces and the corresponding priority inversion *between* different applications.

In SLOTH, mixed-criticality systems can be implemented using priority partitioning or priority placement depending solely on the control flow's *semantic* priority.

e) Difficult integration of real-time modules
Since sophisticated embedded real-time devices are developed in parallel by different module specialists and since the modules have to be integrated for the final system, the system integrator's job is made more complicated by different control flow priority spaces used by the modules. Thus, he does not only have to integrate the modules themselves but also implicit timing constraints in the scattered priority allocations.

In SLOTH, priority integration of real-time modules is facilitated by a unified priority space to reason within.

f) Hampered real-time analysis
Due to the effect of rate-monotonic priority inversion caused by ISRs on the execution and jitter of high-priority tasks, the scheduling analysis and the application of formal workload models on the whole real-time system is hampered and—if feasible at all—has to be more pessimistic than actually required [JS93]. Many theoretical scheduling analysis models work on systems with a single priority space [FMAN06b] or even assume negligible execution times of ISRs [FMAN06b], which is far from realistic for real-world systems. Due to separated priority spaces and occasional synchronization in critical sections, it is therefore very difficult to reason on such models to derive real-time properties such as blocking terms per task [RRWPL03].

SLOTH provides a flattened scheduling hierarchy [RRWPL03] with a unified priority space, which makes real-time analyses simpler.

g) Susceptibility to interrupt overload
Due to the effect of rate-monotonic priority inversion on high-priority tasks, those tasks can be starved by bursty interrupts or periods of interrupt overload, producing livelock situations in network systems, for instance [MR96]. Whereas research has spent a lot of effort on

providing isolation between tasks, little has been done on shielding tasks from interrupt overload to provide strong real-time guarantees [RD05].

SLOTH can provide strong progress guarantees to task level processing by limiting IRQ processing (e.g., by using advanced scheduling techniques such as the sporadic server [Liu00]), shielding the system from interrupt overload.

h) Priority inversion due to synchronization
The effect of rate-monotonic priority inversion is amplified by application tasks that disable IRQs to synchronize with ISRs [FMAN06a]. Since this is a common way for an application to synchronize across those two control flow *types*, high-priority tasks are further delayed while low-priority tasks synchronize with low-priority ISRs in a critical section—even if the low-priority ISRs are not currently running. Thus, critical sections in the application can nullify efforts to reduce or eliminate interrupt locks in the operating system by using non-blocking synchronization techniques.

In SLOTH, the application does not need to disable interrupts for synchronization purposes but can synchronize with all control flows using a stack-based priority ceiling protocol.

2. Synchronization problems

a) Complicated or infeasible synchronization in the application and the operating system
On the application level, synchronization between control flows of different types for mutual exclusion is often complicated or even completely infeasible. Depending on the variety of interactions between different control flow types, the complexity of the synchronization and its mechanisms is increased, making design errors—with a negative impact on system reliability—more probable. Non-preemptive event-driven tasks, for instance, are implicitly executed synchronized to each other, since they are run strictly sequentially. Preemptive event-driven tasks, on the other hand, can potentially preempt each other, requiring synchronization for mutual exclusion. As an additional example, an application task and an application ISR exhibit an asymmetric preemption relationship (an ISR can preempt a task but not vice versa) and are therefore more complicated to synchronize in the operating system [RRWPL03]. The OSEK OS specification, for instance, recognizes this fact and states that the participation of ISRs in the priority ceiling protocol for tasks in critical sections (employed to avoid deadlocks and unbounded priority inversion) is *optional* in any implementation of the standard [OSE05, p. 32]. The same problem

applies to *system-internal* synchronization—that is, the way that the operating system keeps its data structures synchronized with the multitude of available application control flow types and their particular properties.

SLOTH has one single synchronization model and one single synchronization mechanism for the application code and module code as well as for the operating system itself to use—for *all* types of control flows, making synchronization across control flow *types* simpler for the application and its analyses.

b) Broadband synchronization effects
Different control flow spaces also lead to broadband synchronization effects, which might not always correspond to the intended application semantics. System calls are often designed to suspend execution of a whole type space of control flows, such as all category-2 ISRs or *all* ISRs (including category-1 ISRs). These mechanisms categorically affect *all* control flows of a given type, whereas the application might want to distinguish between low-priority and high-priority control flows *inside* that type space for its synchronization purposes.

In SLOTH, all application control flows can flexibly synchronize using only the configured, semantic control flow priorities, enabling arbitrary priority cut-offs.

3. Application development restrictions

a) Complicated usage of control flow types
The availability of a multitude of control flow types and the sometimes subtle differences between them are difficult to grasp and complicated to use for embedded real-time developers, which is underlined by a report on the usage of embedded real-time operating systems [UBM13]. Each control flow type has particular characteristics with respect to performance and latency and particular restrictions for the application developer to respect [RRWPL03].

In SLOTH, there is a single, universal control flow abstraction for the application developer to use; its properties are a simple matter of static configuration by the application developer.

b) Control flow ripping
Due to the variety of non-functional properties of the different control flow types, applications often have to artificially separate their functionality into different control flows of varying types. For instance, since ISRs should be kept short, applications typically have them only pre-process the data, which has then to be communicated explicitly to a follow-up post-processing task.

In SLOTH, a single, properly configured control flow can be used to implement a certain piece of application functionality, leaving the possibility for the separation into different-priority parts solely up to the intended application semantics.

c) One control flow property implies other properties
The control flow semantics of an event handler—run-to-completion or blocking—as well as its priority relative to other event handlers is implied by the event source—software or hardware—but should be defined by the application requirements. The mere fact that a control flow is triggered by a hardware device makes it have a higher priority than any software-activated task in the system. The same applies to interruptions by hardware events like timer IRQs, even if they activate low-priority tasks.

SLOTH offers configurable semantics for its control flows, based solely on the requirements by the application.

d) Unavailable property combinations in control flows
Since there are several properties that control flow types differ in, there are combinations of those properties that are not reflected by any of the types offered by the operating system. Such application control flow requirements are then either infeasible or can only be implemented by a workaround using existing types. Examples for control flows with infeasible properties are software-activated tasks with a higher priority than hardware-activated ISRs; in traditional systems, activation by hardware *implies* that the corresponding control flow has a higher priority than all software-activated control flows. An example for control flow types that need a workaround are blocking ISRs: ISRs can usually *not* be blocked in their execution, which is why a blocking ISR can only be emulated by an ISR that immediately activates a task—which *can* be blocked in its execution.

SLOTH offers a universal control flow abstraction that combines previously separated properties and that can be configured to the desired application semantics, independent of its activation source.

e) Complicated control flow type access rights
The multitude of available control flow types also makes access rights complicated. Operating systems allow some types to invoke all operating system services, others can only call a subset, whereas the rest is not allowed to use operating system services at all; these access restrictions can then be checked by the system either statically or dynamically. Operating systems thus provide large tables and text paragraphs describing those type-specific access rights (e.g., see [OSE05; AUT13]), rendering application development prone to errors related to access restrictions.

In SLOTH, any control flow can use any system service, since it features a unified control flow abstraction.

4. Application evolution inflexibilities

a) Inflexibility during application evolution
If a control flow type is to be changed during the development of an application, this has severe effects on its implementation and integration in the system, rendering system evolution complicated [RRWPL03]. If a control flow previously implemented as a task synchronizes by accessing a blocking lock and is then "ported" to an ISR context, for instance, this will break the system [KE95]. Transitive calls between different code modules further complicate the analysis as to whether it is legal or not for a specific piece of code to be called from a new context. Thus, a decision in favor of one control flow type will be hard to revise during the development or maintenance phases as the application develops since the migration to another control flow type implies lots of unexpected changes in execution semantics as a side effect.

In SLOTH, there is only one mechanism for synchronization and communication between control flows, rendering application evolution simpler.

b) Hampered code modularization
Since the synchronization primitives to use depend on the control flow type context, modular code cannot be oblivious of the context it is called from [KE95; RRWPL03]. Thus, modules have to check for the calling context or even provide different application programming interfaces (APIs) and implementations to accommodate for the correct synchronization mechanism.

Since SLOTH applications feature only unified control flows, module code is always executed in the same context and can therefore be executed unaltered.

All of the listed restrictions and problems are due to the multitude of available control flow types in current embedded operating system designs. As briefly mentioned with each restriction, the SLOTH operating system presented in the rest of this thesis can overcome those problems through its hardware-centric design and flexible hardware abstraction (see Section 3.2 and Section 3.3). Due to its hardware-centric basis, SLOTH can assert its very good non-functional properties to the embedded real-time application running above if the hardware platform is capable to support SLOTH's universal control flow abstraction natively; otherwise, it has to resort to emulating parts in software at the expense of impaired non-functional properties.

2.3 Missed Optimization Potential in Non-Functional Properties of the Operating System

The fact that embedded operating systems offer a multitude of control flows to reflect different non-functional requirements by the application shows the importance of non-functional properties for embedded applications. This is especially true for embedded *real-time* applications, which have strict requirements on the response times in their physical environment. In addition to timely computation results, some real-time systems also have demands for high performance and high-resolution timing and scheduling of their control flows, such as applications that include video processing.

2.3.1 Influence of Operating Systems on the Non-Functional Properties of an Embedded System

Operating systems, whose only task is to support applications in fulfilling their requirements, are therefore also developed with a focus on non-functional properties. Since operating systems provide no business value of their own, they are subject to particular scrutiny with respect to their influence on the non-functional properties of the whole embedded system. According to a recent study on embedded systems by EETimes, a significant percentage of embedded software engineers would want an improved operating system, and the main reasons *not* to use an operating system at all are that they consume too much memory and too much processing power [UBM13]. A real-time operating system affects the embedded system in several ways:

CPU utilization
: The processing overhead introduced by the operating system reduces the theoretical CPU utilization that is available for the real-time application. The context switch time is especially important for systems with high activity and a high context switch rate, since context switches can then impose a high overall overhead on the system. In real-time systems, the CPU overhead induced by the operating system hampers the schedulability analysis (and other model reasoning) and introduces jitter, which is problematic for equidistant sampling applications. In systems with sustained overload and highly loaded schedules, only the reduction of system overhead can lead to a schedulable system [RRWPL03].

Event latencies
: Real-time applications have to assert certain response times—which are extremely low for applications like engine controls—to their environment for input events. The interrupt latencies for those events are influenced by the hardware, but also to a major extent by the operating system, which preprocesses interrupts, adding dispatch latencies, and—depending on its in-

ternal synchronization (see Section 2.2.1)—might disable them in system-internal critical sections, again increasing the overall event latency.

Memory overhead
The operating system code also introduces overhead in code and data memory of the embedded system. Since mass-produced embedded systems have to optimize for low-memory microcontroller derivatives, the overhead induced by the operating system can play a major role in that respect. Additionally, the cache load induced by the operating system in the system instruction and data caches can lead to additional response time jitter, decreased performance, and increased latencies.

Energy efficiency
Since the operating system imposes computational overhead, it reduces the sleep cycle times of the embedded system. This has a negative impact on the energy efficiency, which is especially important for battery-operated *mobile* embedded systems, but which also affects the overall gas mileage of an automobile featuring dozens of those embedded systems, for instance.

One main goal in the design of an embedded real-time operating system should therefore be the optimization of system-induced latencies and overhead.

2.3.2 Problems Introduced by Hardware Abstraction Layers

However, traditional operating systems feature one design element that stands in the way of unrestricted optimization: the hardware abstraction layer, which an operating system engineer usually defines for his operating system. Operating systems were introduced as abstract machines with a standard interface to increase the portability of applications across hardware platforms; to increase the portability of the operating system *itself*, a system-internal HAL is defined within the operating system, which encapsulates and hides hardware platform features from the rest of the operating system implementation by presenting a static, non-configurable interface. For the sake of portability of the operating system, and therefore for the *operating system engineer*, such an abstraction layer is very useful, since porting the operating system implementation to another hardware platform will then be restricted to re-implementing the HAL. However, the abstraction also hides hardware features that are beneficial to the operating system and, more importantly, to the application—although those features might not be available on *all* supported platforms. Additionally, some concepts presented by a HAL do not map well on certain hardware platforms, forcing the operating system to emulate those concepts in software then; the application, however, will not be aware of the increased latency. Specifying the location of a HAL when designing an operating system from scratch will always be subject to a trade-off decision between efficiency and portability of the underlying operating system implementation [CMMS79]. Beneficial hardware features mostly have

an impact on the *non-functional* properties of the operating system; by abstracting from them, potential for optimization of those properties is missed, since the operating system has to implement those features in software, impacting the non-functional properties passed to the application in a negative way.

Consider, for instance, the fact that most modern microcontrollers feature an interrupt controller with support for many interrupt priorities and interrupt preemption. Since previously, most platforms only had a single interrupt level at their disposal, many operating system HALs abstract from that property and do not make use of it to implement prioritized control flows—and instead have to implement priority management in the operating system software. This will, however, increase scheduling and dispatching latencies, and has a negative impact on the operating system's non-functional properties as perceived by the application.

A second example is the multiplexing of a single timer interrupt to satisfy the requests by different application timers as performed by any standard operating system design in its HAL. Several microcontroller platforms, however, feature sophisticated timer arrays that can implement several timers simultaneously and directly in hardware, avoiding the use of multiplexing software and making software abstractions built upon a single timer obsolete.

Both examples show that those problems often stem from the fact that microcontroller hardware has evolved to include more sophisticated features and implementation possibilities (see also the study [UBM13] on embedded systems, which shows that 32-bit microcontroller use is spreading), whereas most hardware abstraction layers and their interfaces have not evolved at all. Additionally, embedded hardware platforms are diverse enough as they are to make a common HAL interface within an operating system inappropriate or at least inefficient. Other researchers have also recognized that traditional real-time operating systems that specify high and standardized HALs prevent innovations from reaching the operating system implementation above [Frö01] and that the missing analysis of hardware peculiarities is a major problem in the design of real-time systems [Ste99] (see discussion of work related to operating system tailoring in Section 7.3.2). Thus, applications running on top of such operating systems cannot make use of specialized hardware features; for some applications, this might even mean missing their non-functional requirements although the hardware platform could satisfy them if it were used properly by the operating system. Therefore, facilitating the portability of an embedded operating system by using a static HAL design comes at the price of reduced efficiency, which itself has a negative effect on the perceived non-functional properties of the operating system and therefore on the application.

Note again that these considerations are subject to a trade-off decision between portability and efficiency-related non-functional properties. In other domains, portability remains the most important property; the operating system abstraction layer OSAL developed by NASA, for instance, even abstracts from standardized operating system interfaces such as POSIX to further increase porta-

bility of the applications [NAS13]—at the expense of further missed optimization potential. This thesis uses the automotive standards OSEK OS, OSEKtime OS, and AUTOSAR OS as examples for *external* operating system interfaces to provide an unbiased basis for discussion of the SLOTH approach and to have a fair basis for evaluation against other implementations of those standards. *Internally*, however, it challenges the location of the hardware abstraction interface.

2.3.3 SLOTH: Hardware-Centric Operating System Design for Latency Hiding

The SLOTH approach developed and presented in the rest of this thesis proposes to raise the system-internal hardware abstraction interface. The SLOTH goal is for the operating system implementation to use innovative and special hardware features to benefit the application in the end; SLOTH does not use abstraction for the purpose of abstracting away *beneficial* hardware features. In particular, SLOTH features concepts for latency hiding with respect to control flow scheduling and dispatching by design.

In consequence, SLOTH can mitigate the problems introduced by traditional hardware abstraction as discussed before. SLOTH provides an execution environment with very low execution overheads, making applications feasible that are not schedulable in other systems with sustained overload and making the application of real-time analysis models feasible; it reduces event latencies, providing low application response times; it limits memory overhead, making the use of smaller microcontroller derivatives feasible; and it increases energy efficiency by lower execution overhead and fewer system interrupts.

2.4 Problem Summary

Current embedded real-time operating systems offer a multitude of control flows, trying to find the best possible mapping from non-functional requirements by the application to non-functional properties offered by the hardware. Additionally, they employ a static hardware abstraction layer in their implementations to reduce the porting effort in the operating system implementations themselves.

This approach, however, leads to a number of problems for the real-time application developer, ranging from inflexibilities and restrictions to system predictability problems. Furthermore, current operating system designs purposefully accept that they miss out on potential for optimization in their implementations, which could be directly handed to the applications to optimize *their* non-functional properties.

3 The SLOTH Approach and SLOTH Design Principles

> Brian: Sometimes, taking things slower is better. Just ask any SLOTH!
> *Scene cut.*
> Narrator talking to a SLOTH: Hey, is sometimes taking things slower better?
> *Really long reaction pause with* SLOTH*'s eyes moving really slowly toward the camera.*
> SLOTH: Yah.
>
> *From the TV series* Family Guy*, episode S10E11, 2012*

The analysis in the previous chapter has identified and detailed the problems for applications using state-of-the-art real-time operating systems; the main limitations affect their deterministic real-time execution and missed optimization opportunities. In order to mitigate or even eliminate those problems, the following sections present the main parts of the SLOTH approach and the SLOTH operating system design principles. These entail

- two-dimensional operating system tailoring (see Section 3.1),
- hardware-centric operating system design for the purpose of latency hiding (see Section 3.2)—with special designs for control flows (see Section 3.2.1) and timing facilities (see Section 3.2.2)—
- and flexible hardware abstraction (see Section 3.3).

All of those three principles are related to the hardware-centric nature of the SLOTH approach, and all of those three principles depend on each other. This chapter details these basic principles before introducing the SLOTH operating system design in Chapter 4.

3.1 Two-Dimensional Operating System Tailoring

Traditional operating systems provide full-blown functionality to their applications to accommodate all possible use cases at run time. Traditional *embedded*

operating systems try to reduce their system footprint in the overall embedded system by providing a configurable operating system base that is tailored to the application needs. This state-of-the-art *upward adaptability* to the application can already improve most non-functional properties of the operating system.

In the SLOTH operating system approach, I extend this approach by providing *downward adaptability* of the operating system by configuring it to the hardware properties. Tailoring an operating system downward toward the hardware does not only mean re-implementing its hardware abstraction layer, but rather re-thinking how mechanisms offered by the hardware can be comprehensively used in the operating system implementation. The goal is to provide an operating system that is simultaneously adaptable in the two dimensions of applications above *and* the hardware below to achieve unprecedented improvements in the non-functional operating system properties.

In a first step, the two-dimensional tailoring approach needs detailed information about the application and its configuration. Thus, it has to analyze the application and system configuration that the application provides and has to scrutinize it for information that can be deduced from it. This way, SLOTH can leverage static application knowledge for system optimization. The goal of this process is to use partial evaluation in order to move functionality and data from dynamic execution and evaluation at run time to static generation and evaluation at initialization time. By executing functionality at run time only once when the system boots and initializes, SLOTH pursues the goal to avoid *actual* run time costs during the system's productive phase *after* the initialization as much as possible.

In a second step, a generative approach is taken to adapt the system software as well as possible. By generating many operating system parts depending on the information gained by the analysis step, exactly fitted operating system source code can be produced; the generation process is augmented by both architecture-specific and application-specific code generation rules. The compiled operating system will then be the ideal match for both the application-specific requirements and the architecture-specific properties. This way, the tailored operating system will have non-functional properties that are optimized in both dimensions that an operating system interacts with.

Other operating system tailoring approaches include synthesizing optimized system calls at run time (Synthesis [PMI88; MP89]), using dedicated processors for communication and interrupts (PEACE [BGSP94]), improving local inter-process communication costs in microkernels (L3 [Lie93; Lie95; Lie96]), and exokernel designs [EKO95], which are highly hardware-specific. Most of those approaches are only partly applicable to systems for the embedded real-time domain—which is the focus of this thesis—since they assume different hardware platforms (desktop or server platforms with virtual memory) or bear other non-functional disadvantages that are relevant to the domain such as increased jitter at run time. These systems are comprehensively discussed and put in relation to the SLOTH approach in Section 7.3.2.

3.2 Hardware-Centric Operating System Design for Latency Hiding

The main SLOTH principle is to develop a design for an operating system that is hardware-centric in its nature, using functional and architectural properties of the underlying platform instead of blindly abstracting from potentially beneficial hardware peculiarities to opt for portability at all costs. The SLOTH design aims at focusing on positive hardware particularities and properties and tries to "hand up" those features to the operating system implementation and the application in order to provide them with the ability to optimize their non-functional properties. SLOTH tries to bridge the semantic gap between the operating system interface and the hardware platform by adapting to the hardware instead of abstracting from it. This design philosophy deliberately refrains from making portability of the operating system a top priority; nevertheless, the approach is not platform-specific and makes porting easy by still defining a clear hardware abstraction interface, which can be implemented by directly using hardware features on some hardware platforms (see also Section 3.3).

To the application, the SLOTH operating system is a logical extension of the hardware platform, offering abstractions that embrace hardware peculiarities in their implementations and potentially in their interfaces to the application. This vertical propagation of hardware properties does not necessarily have to have syntactic effects on the application interface; the SLOTH operating system prototype described in the rest of this thesis, for instance, presents a standardized OSEK OS interface to the application as an example. If the platform allows for such a hardware-based design, the SLOTH implementation of the example OSEK OS API can skip most software parts completely and directly access hardware services instead. On platforms with less powerful abstractions, a more comprehensive operating system software layer might be needed; in any case, the operating system implementation remains transparent to the application. In general, the SLOTH design aims at implementing interfaces by abstracting downward to determine which mechanism is the most efficient one to implement a given interface.

As an example for interfaces that can be altered due to the hardware-centric SLOTH design, consider restrictions in the cardinalities of operating system objects. SLOTH can offer to reduce the universality and generality of its implementation by restricting the application to n control flows or m time-triggered task activations, for instance. In this case, it can optimize for a special case that is well supported by the hardware, with the corresponding positive effects on the non-functional properties of the whole system. That way, SLOTH can optimally support a certain class of application while offering a fall-back option with more software effort for all other applications.

The overall goal in centering the operating system design on the hardware platform is to hide system-induced latencies where possible. By using hardware-specific features more efficiently, latencies perceived by the application while ac-

cessing the operating system—and, indirectly, the hardware—can be reduced. The design, however, has to respect the fact that setting up and configuring the corresponding hardware subsystems using appropriate machine instructions might at first *increase* the corresponding system latency, at least during the initialization phase. Thus, these additional latencies need to be considered and kept as low as possible to keep the whole approach worthwhile.

The hardware-centric SLOTH approach is particularly applicable to special-purpose embedded systems in domains with special requirements. In those domains, focusing on the hardware features is especially worthwhile due to the availability of a wide variety of microcontrollers with different functionality. A recent study shows that the variety of data path widths in embedded systems still ranges from 8-bit to 64-bit controllers, which therefore offer a wide variety of properties [UBM13]. Additionally, the study shows that the use of more powerful 32-bit microcontrollers is spreading, making the SLOTH approach more worthwhile, since new hardware functionality is seldom reflected in the operating system interfaces nor used in the operating system implementation. However, the approach is also interesting for general-purpose operating systems: By using new instructions developed by chip manufacturers such as Intel or AMD to build more efficient operating system abstractions, the non-functional properties of the operating system and the application can also be improved. The concrete advantages of the SLOTH approach depend on the actual hardware platform and its available features, however.

In order to fully exploit the hardware and its capabilities, the SLOTH approach requires constant feedback loops during the implementation to measure the corresponding non-functional properties such as latencies, performance, and code and data size of the generated operating system. In this way, the SLOTH development is similar to the way the L3 microkernel was developed [Lie93]: All design and implementation decisions require discussions about the implications for the non-functional properties. This entails taking close looks both at the high-level code generated by the SLOTH generator, which tailors the operating system to the application and the hardware, and at the machine instructions generated by the corresponding back-end compiler; this way, bottlenecks can be discovered and addressed (see also discussion of related work in Section 7.3).[1]

Thus, the SLOTH design approach is a hardware-driven approach and is based on scrutinizing the target platforms. Only after common *and* special hardware features have been identified is the operating system designed in order to benefit from these specialties as much as possible. If the hardware offers powerful abstractions, SLOTH implements its abstractions in hardware using only a very thin operating system software layer; for such an ideal implementation that yields very beneficial non-functional properties, consider Section 5.4 for the cor-

[1] Note that the SLOTH operating system prototype is programmed mostly in C, whereas L3 uses assembly at least in its critical parts. All of the critical parts in SLOTH, however, have been examined instruction by instruction in assembly to check whether the C compiler applied optimizations in a satisfactory way as would have been possible by directly programming in assembly.

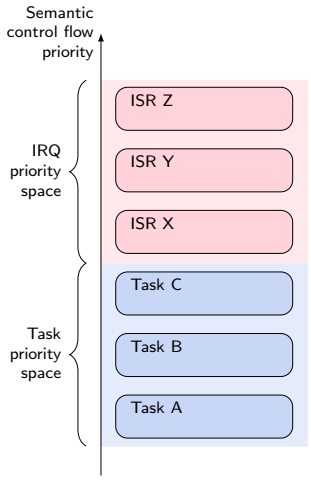
responding abstract hardware platform requirements. If the hardware offers less powerful abstractions, SLOTH's hardware-centric design makes it implement its abstractions using more operating system software. SLOTH's overall goal is therefore to make the application benefit from the hardware properties *as much as possible* with the given hardware platform. The rest of this thesis only describes the fall-back to software emulation in the operating system when it is of particular interest; otherwise, a software implementation is similar to the one of other operating system designs. The following two subsections presents the two main instances of the hardware-centric design approach, which target the design of control flows and operating system timing features in a hardware-centric way.

3.2.1 Hardware-Centric Control Flow Design

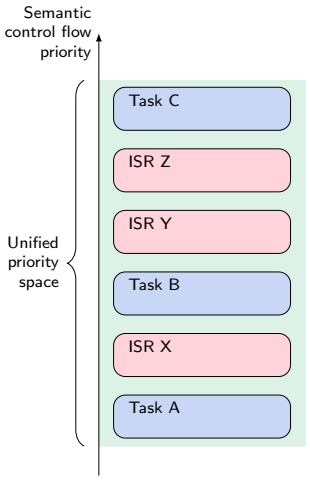
The first instance of the hardware-centric SLOTH design approach targets system control flows. SLOTH views control flows as first-class abstractions of the hardware platform and focuses on using them to their full potential. The SLOTH operating system therefore embraces the control flow particularities offered by the hardware and presents to the application a universal control flow abstraction—the central SLOTH abstraction. If the hardware platform permits, it maps most control flow abstractions offered by typical embedded systems directly to an implementation in an interrupt service routine—the most basic control flow abstraction of a hardware platform. This way, control flows are unified and run in a unified priority space, allowing for arbitrary priority distributions depending solely on the *semantic* priorities as specified by the application (see Figure 3.1, adapted from [Liu00, p. 505]). Additionally, this measure effectively out-sources the scheduling and dispatching of all kinds of application tasks to the hardware interrupt controller with its interrupt priority space, which is wired to do this in hardware. For such an ideal implementation of SLOTH's universal control flow abstraction, the hardware platform needs to be powerful enough for a given application; an application with n tasks and m ISRs, for instance, would ideally require an interrupt controller with $n + m$ IRQ sources and priorities. Again, if parts of the requirements for an ideal implementation of operating system parts in hardware cannot be fulfilled by the underlying hardware platform, the hardware-centric SLOTH approach aims at emulating the missing parts in software to adapt to the platform.

Note that in a simple co-operative system, control flows can also be executed in a single priority space; if the application features either only ISRs or only tasks, its control flows run only in the ISR priority space or only in the task priority space, respectively. SLOTH's control flow model also allows for such systems, but additionally allows for more challenging application designs with preemptive tasks and ISRs that are run in a single priority space.

In addition to using the interrupt controller and its priorities for the design of the control flow abstractions themselves, the SLOTH approach also focuses on using the interrupt subsystem for the design of its corresponding system calls. If



(a) Mandatory priority distribution in a traditional preemptive operating system (adapted from [Liu00]).



(b) Possibility for a priority distribution in the preemptive SLOTH operating system.

Figure 3.1: Example priority distributions of preemptive control flows in a traditional operating system and in the SLOTH operating system. In a traditional preemptive operating system, ISRs have to have a higher priority than all tasks (a). In SLOTH, the semantic priorities can be distributed arbitrarily between control flows, independent of their types (b).

made possible by the hardware platform, control flow system calls are therefore designed using the explicit or implicit alteration of the current interrupt priority using hardware-specific instructions.

By these two means, most real-time problems, restrictions, and inflexibilities discussed in Chapter 2 can be mitigated or even eliminated. Additionally, the system performance is optimized by hiding the latency for scheduling and dispatching the application control flows in the interrupt system (see evaluation in Chapter 6).

Other approaches to unify priority spaces are presented by Leyva-del-Foyo et al. [FMAN06a; FMAN12], Dodiu et al. [DGG10], Zhang and West [ZW06], and Lee et al. [LLSSHS10], amongst others. These approaches try to limit the problem of rate-monotonic priority inversion by either letting ISRs run as tasks that are signaled by different kinds of IRQ stubs—using only the task priority space—or by letting tasks run on interrupt level by setting it upon each task dispatch—using only the interrupt priority space. In contrast to the SLOTH approach, the first kind of approach still exhibits rate-monotonic priority inversion during the shortened execution periods of the IRQ stubs. The second kind of approach does not allow low-priority ISRs to interrupt high-priority tasks, but it does not dispatch a high-priority task when it is activated during the execution of a low-priority ISR. To

reach this desired behavior, the software task scheduler has to be further modified to respect both the task priority *and* the interrupt priority in its decisions, preempting an ISR by a task when necessary. Compared to the SLOTH approach, *both* kinds of related approaches have the disadvantage of an increased operating system overhead due to the manual adaptation of the IRQ masks or the IRQ level upon each task dispatch, due to having to keep it consistent with the software task priority, and due to the missing information about ISR control flows in the task scheduler, which can lead to unnecessary context switches in some cases. In SLOTH, the system performance is optimized by hiding latencies in the hardware interrupt subsystem. All of the approaches related to rate-monotonic priority inversion and to the proposal of interrupt thread abstractions are discussed in Section 7.3.1.

3.2.2 Hardware-Centric Timing Design

The second design instance that focuses on hardware-centric means is the way that the SLOTH design approaches timing services. In time-triggered real-time systems that schedule and dispatch tasks in equal-length frames computed offline (see also Section 1.5), a single timer as available on any hardware platform suffices to support a time-triggered operating system design for a cyclic executive. This executive is invoked in regular intervals—the frame length—and monitors the released tasks for the current frame and potentially overrun tasks from the previous frame. If a second timer with a corresponding counter register is available, the hardware-centric SLOTH design can use it for an efficient design of slack stealing: The second counter acts as a counter of the remaining slack time in the frame, preempting aperiodic and sporadic tasks if necessary. If only a single timer is available, the design multiplexes it for the regular frame interrupt and the interrupt for exhausted slack time.

In time-triggered systems that do not employ frames but use statically computed dispatcher rounds, the operating system has to activate control flows based on pre-configured points in time. Instead of multiplexing those points in time using a single hardware timer that any hardware platform offers, the SLOTH approach focuses on using timer *arrays* with multiple cells as available for instance on some of the more capable microcontroller platforms. If the hardware features a timer array, SLOTH directly maps operating system timer abstractions to hardware timer cells where possible. The goal of this process is to move timer reconfiguration code at run time to statically generated code at initialization time in order to avoid run time costs during the system's productive phase as much as possible, leading to little or no logic for the dispatcher to cause overhead at run time.

Again, this SLOTH design principle adapts to hardware platforms that feature enough timer cells for a given application—if so, the application is given an optimized operating system implementation to improve its non-functional properties. This potential for optimization is also due to the development of the hardware;

microcontrollers have evolved since when timers were rare peripherals, and the SLOTH approach focuses its adaptive and hardware-centric design around that fact to benefit the application where possible.

Previous work on operating system timers also focuses on their efficient and predictable implementation and the corresponding analysis of implementation alternatives. Work by Varghese and Lauck [VL87], Zhou et al. [ZSR98], and Fröhlich et al. [FGS11], amongst others, proposes several algorithmic and scheduling improvements in the timer part of the operating system software and the use of specialized timer hardware. The SLOTH approach, in contrast, focuses on commodity off-the-shelf hardware and how to provide a hardware-centric timing design for such platforms to improve timer efficiency and accuracy; a detailed discussion of the relation to other approaches can be found in Section 7.3.4.

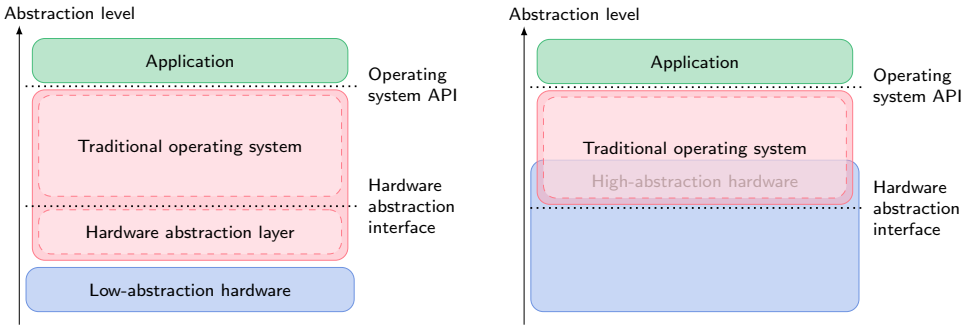
3.3 Flexible Hardware Abstraction

Originally, applications were developed specifically for a given hardware platform and had to be ported to run on a different architecture. Operating systems were then introduced to provide an architecture-independent abstract machine interface to the application in order to mitigate the application porting efforts. In order to facilitate porting of the operating system itself, operating system vendors use a system-defined hardware abstraction layer to encapsulate all architecture-specific parts of the operating system so that only that capsule has to be ported for a different hardware platform (see Figure 3.2(a)).

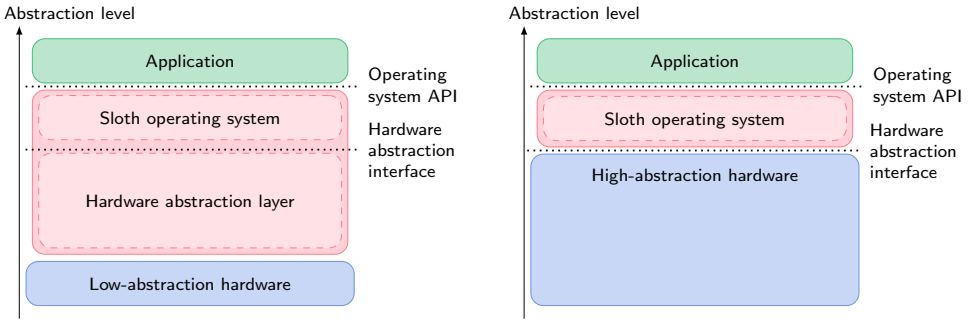
Due to the problem of missed optimization potential because of a fixed and inflexible hardware abstraction layer (see Section 2.3), SLOTH uses a different approach that employs *flexible* hardware abstraction internally. SLOTH hardware abstraction features a high-level hardware abstraction *interface* together with a flexible hardware abstraction *layer* (HAL)—the interface *implementation*. The combination of those two approaches allows for the operating system to be flexibly and comprehensively optimized for the target hardware platform.

3.3.1 High-Level Hardware Abstraction Interface

Current operating system designs feature an internal hardware abstraction interface that is the least common multiple of available hardware platforms and is therefore located at the lowest level possible (see Figure 3.2(a)). This low interface is the root for hindering cross-interface optimization on hardware platforms that offer more powerful abstractions; in the right part of Figure 3.2(a), for instance, the traditional operating system software cannot make use of the highest-level hardware abstractions, rendering corresponding operating system optimizations infeasible (see overlap of operating system abstractions with hardware abstractions). Thus, SLOTH claims a high-level hardware abstraction interface that expresses the semantics of high-level abstractions as an abstract machine with defined pre-conditions.



(a) When traditional operating systems are used for hardware with high-level abstractions (right), the operating system software cannot make use of those high-level abstractions due to the low hardware abstraction interface.



(b) When a SLOTH operating system is used for hardware with high-level abstractions (right), the SLOTH operating system software can directly make use of those high-level abstractions due to its higher hardware abstraction interface.

Figure 3.2: Hardware abstraction approaches in traditional operating systems and in the SLOTH operating system. Traditional operating systems with a low-level hardware abstraction interface limit the operating system flexibility by preventing the use of high-level hardware abstractions due to the infeasibility of cross-interface optimizations (a). With the SLOTH approach of a high-level hardware abstraction interface, the SLOTH operating system software can directly use high-level hardware abstractions on suitable platforms, serving as a simple and thin mapping layer (b).

This approach allows for high-level optimizations since the implementing software—the HAL—has an understanding of the operation that is being performed by an interface function and can therefore find an efficient and hardware-specific implementation. On less powerful platforms, this hardware-specific implementation uses simpler, low-level hardware abstractions together with rich operating system software that implements the high-level abstractions—similar to traditional operating systems (see left part of Figure 3.2(b)). On more powerful platforms, however, the hardware-specific implementation can directly make optimal use of more sophisticated, high-level hardware abstractions, applying

only a very thin and efficient layer of operating system software if any (see right part of Figure 3.2(b)). Thus, the SLOTH hardware abstraction interface allows for different actual locations of the hardware–software boundary.

This extra flexibility allows the SLOTH operating system to provide more efficient and optimized HALs that benefit the non-functional properties of the operating system, and, therefore, also the application itself. The higher level of the hardware abstraction interface effectively means that the HAL implementation provides more powerful abstractions to the software part of the operating system, making that part a thinner layer of software. Note that despite its higher level, the interface is still clear and fixed as in traditional systems in order to allow for straight-forward portability of the operating system. *Below* the hardware abstraction interface, a SLOTH operating system for a given hardware platform can also have different software–hardware traits for different system calls *internally*; one system call might be implemented by directly using an appropriate hardware abstraction, whereas a different one has to be implemented completely in software in order to offer some compatible interface to the application. The high-level hardware abstraction interface only provides *the ability* to make use of more sophisticated hardware abstractions if the platform permits.

3.3.2 Flexible Hardware Abstraction Layer

In current operating system designs, the implementation of the hardware abstraction interface—the HAL—for a given hardware platform is fixed. Traditional HALs only provide limited adaptation possibilities through the use of configuration variables to include or exclude HAL components, but the HAL component code itself is static. This design opts for generality in its implementation to accommodate for any application, but it sacrifices optimization possibilities to reach higher levels of efficiency by flexibly adapting to the application.

The SLOTH hardware abstraction approach provides additional flexibility by adapting its hardware abstraction layer to the application as the rest of the operating system. SLOTH therefore features a *flexible* hardware abstraction layer and tailors its HAL components to the application abstractions as specified by the application configuration (see Figure 3.3). This application specialization is part of the SLOTH HAL concept and has to be performed in a step prior to the system building, similar to the way traditional operating system tailoring to the application is implemented.

The flexible part of the SLOTH HAL is a non-static part that needs to be generated depending on the application and the hardware (see Figure 3.3). The flexible meta HAL includes code for all architectures and potential applications and therefore first needs to be specialized for the selected hardware platform to output a flexible HAL instance. In a second step, the HAL build step additionally takes as its input the application configuration with its defined application abstractions. This input allows for the flexible HAL instance to use generative optimization techniques in order to tailor its implementation to the application

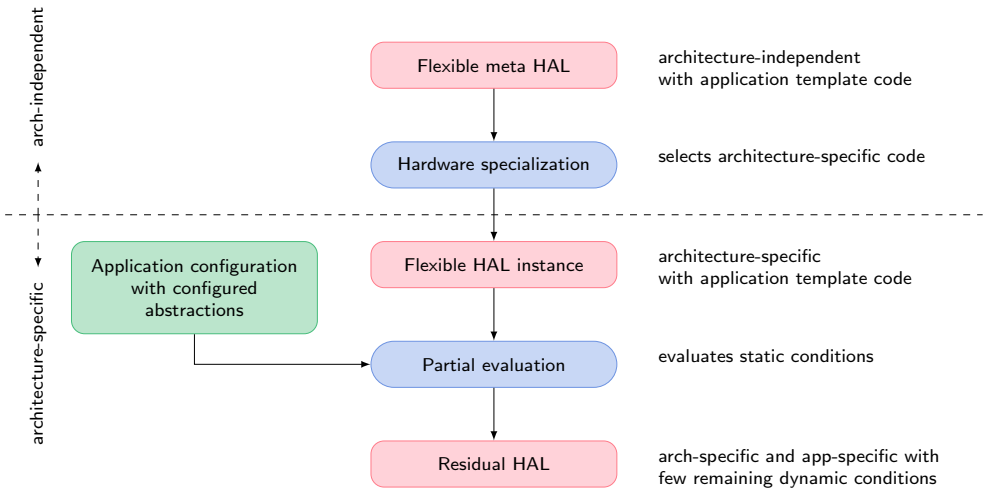


Figure 3.3: SLOTH flexible HAL and its tailoring steps. SLOTH provides a flexible meta HAL, which includes template meta code to produce a flexible HAL instance. This flexible HAL instance is then partially evaluated to generate the residual HAL, which is actually deployed in the specialized system.

requirements. Additionally, by evaluating the static application configuration, the SLOTH flexible HAL instance can use partial evaluation in order to move configuration-dependent code and data from dynamic evaluation and execution at run time to static evaluation and generation at compile time. This approach is especially well applicable to statically configured systems, which most embedded real-time systems—which are the main target of the SLOTH prototype implementation and of the discussions in this thesis—are (see also Section 1.5). The output of the partial evaluation is a residual HAL program that is specialized to both the hardware *and* the application. Since the residual HAL program had its configuration conditions evaluated statically as far as possible, it can eliminate superfluous dynamic run time checks, rendering its implementation as thin as possible. Similar optimization mechanisms with encouraging results have been used in the Synthesis and Synthetix approaches, for instance [PMI88; MP89; PABCCIKWZ95]; the corresponding operating systems feature system specialization at run time to tailor the system call implementations to the application. The SLOTH approach moves this tailoring step to compile time by *statically* generating parts of the system and the HAL, since—unlike with Synthesis and Synthetix—the target domain of embedded real-time systems can usually not tolerate dynamic adaptation mechanisms, and since applications in this domain have static application information available anyway.

The resulting residual HAL implementation is not only application-specific, but also—like any other HAL—architecture-specific, effectively adapting in both dimensions (see also Section 3.1). The flexible part of the SLOTH HAL is archi-

tecture-specific as is the static part; however, the generation approach allows for easy extensions and adaptations and therefore for straight-forward porting.

In combination with the possibility for high-level optimizations through its high-level hardware abstraction interface, this allows for the flexible HAL to become a simple mapping layer that directly maps application abstractions to hardware abstractions where possible. Effectively, the SLOTH flexible HAL is a meta component that is tailored to a residual thin layer providing powerful and efficient high-level abstractions and tuning itself for the target machine and application, exposing advantageous machine features to be made use of in the operating system.

As mentioned in Section 3.1, other work has investigated the role of hardware abstraction in operating systems and how to better tailor an operating system to the underlying platform. The SLOTH flexible hardware abstraction approach is encouraged by these findings to design its flexible hardware abstraction layer; the exact relation of these other pieces of work to SLOTH is discussed in Section 7.3.2.

3.4 Approach Summary

The SLOTH approach to operating system design builds on the hardware as a first-class element for the operating system to consider. In that spirit, it employs two-dimensional tailoring of the operating system to both the application and the hardware, it proposes an operating system design that is hardware-centric in its nature in order to hide latencies in its control flow and timing facilities, and it uses a high-level and flexible hardware abstraction layer inside the operating system. All of those design principles are targeted toward making better use of the hardware to benefit the non-functional properties of the operating system and the application. If the hardware offers specialized features, the SLOTH operating system design adapts to accommodate for those; if the hardware does *not*, the SLOTH design emulates the corresponding elements in software as do other operating system designs.

4 The Design of the SLOTH Operating System

> Thither we drew; and there were persons who were staying in the shadow behind the rock, as one through indolence sets himself to stay. And one of them, who seemed to me weary, was seated, and was clasping his knees, holding his face down low between them. "O sweet my Lord," said I, "look at him who shows himself more indolent than if SLOTH were his sister."
>
> *From the epic poem* Divina Commedia *by Dante Alighieri (translation by Charles Eliot Norton), Purgatorio, Canto IV, 1321*

Following the SLOTH approach and the principles described in the previous chapter, this chapter presents the design of the hardware-centric SLOTH operating system, which employs latency hiding techniques to optimize its non-functional properties on favorable hardware platforms. The SLOTH operating system design can be used in different flavors to accommodate for different real-time use cases:

- The original SLOTH operating system targets event-triggered systems with basic tasks, which run to completion (see design in Section 4.2).
- The SLEEPY SLOTH operating system extends the SLOTH design by the support for extended tasks, which can be blocked during their execution (see design in Section 4.3).
- The SLOTH ON TIME operating system design supports time-triggered systems and systems with time-based activations and features (see design in Section 4.4).

All of the SLOTH flavors share the focus on hardware facilities to be incorporated in the operating system design, which is detailed in the following sections. As mentioned in Section 3.2, the description picks up the fall-back to software emulation where it is of particular interest.

4.1 The Programming Model and the Universal Control Flow Abstraction of the SLOTH Operating Systems

The programming model of the SLOTH operating system is inspired by the interfaces of the OSEK OS, OSEKtime OS, and AUTOSAR OS specifications (see also Section 1.5). Where applicable, the SLOTH system calls have the names and parameter sets as specified by the standards (in order to be able to compare the prototype implementation to other operating system implementations in Chapter 6); additionally, SLOTH offers abstractions and system calls for the application that go beyond and deviate from those standards where appropriate. As common in most embedded systems, SLOTH is configured completely statically—the application developer specifies all the abstractions the application needs from the operating system in advance in an *application configuration* (e.g., tasks, ISRs, resources, alarms, etc., as described in this chapter), together with their static properties, such as their priorities (see also example configuration in Table 4.1). In order to be able to configure the control flow priorities, the application needs to analyze its control flows according to an event-triggered scheduling scheme such as the rate-monotonic or the deadline-monotonic algorithm, for instance; this step is not in the scope of the SLOTH operating system. Application configurations for *time-triggered* SLOTH ON TIME systems (see Section 4.4) mainly entail times for frame lengths and hyperperiods, and mappings of tasks to frames, activation points, and deadlines—as output by an external tool that constructs the time-triggered schedule, for instance. Depending on the support by the underlying hardware platform for the hardware-centric SLOTH design, the configuration also entails hardware-specific information, such as which IRQ source to map which task to.

Additionally, some of SLOTH's *system properties*, which often address trade-off decisions (e.g., increased RAM usage versus higher latencies exhibited by the operating system), are configurable by the application in order to make such decisions depending on its non-functional requirements. Such properties do not only entail system-internal implementation configurations, but also application-visible properties such as to use non-preemptive tasks in order to save allocated RAM for their common stack and to facilitate task level synchronization, for instance. Both the application configuration and the system configuration are supplied in a separate programming entity (see also implementation in Section 5.1).

In all of the SLOTH systems—with stack-based, blocking, or time-triggered execution—the main *internal* control flow abstraction is the interrupt service routine as supplied by the underlying hardware platform. SLOTH forwards this hardware abstraction to the application with software adaptations that are as minimal as possible, depending on the capabilities of the hardware concerning the amount of interrupt sources at its disposal as well as the offered priority space. SLOTH utilizes the interrupt service routine as an application-configurable control flow abstraction and offers tailored *types* depending on the application

requirements, which has a direct effect on the amount of allocated control flow context and on the way it is saved and restored. An extended task—which is allowed to use blocking system calls—as offered by the SLEEPY SLOTH operating system, for instance, has as its property a more sophisticated software prologue to save and restore blocked state (see Section 4.3). Since all of the control flows of a SLOTH application are internally mapped to a single, universal control flow abstraction—the hardware interrupt service routine—it benefits from several non-functional property improvements, which entail lower latencies and the elimination of rate-monotonic priority inversion by design, amongst others (see comprehensive evaluation in Chapter 6).

4.2 SLOTH Systems with Stack-Based Task Execution

The design of the original SLOTH operating system targets event-triggered systems that feature a strictly stack-based task execution pattern.[1] Such a pattern is exhibited when the application only entails basic tasks in its configuration, which run to completion on a single stack, but no extended tasks, which can potentially block and are therefore usually allocated a stack of their own.[2] Such systems have the advantage that they do not need static stack memory allocations *per task*, which themselves need to accommodate for the worst-case stack usage per task.

This kind of operating system and its abstractions are ideally suited to be mapped to interrupt service routines and interrupt controller instrumentation if the interrupt subsystem of the hardware platform permits. The following sections describe the design of those abstractions and how they interact to provide the desired system behavior.

To illustrate the design of the SLOTH operating system, an example event-triggered real-time application is used. The example application features three tasks, one ISR, two resources, and one alarm, with the properties presented in Table 4.1.[3] An example control flow of this application is shown in Figure 4.1; the figure is explained step by step when introducing the corresponding operating system abstractions in the following sections.

[1] This thesis calls an operating system that supports such an execution pattern to be *stack-based*. As in *event-based* systems, tasks cannot be blocked in their execution; however, they can be preempted. In the worst case, stack-based systems need to allocate the common stack to accommodate the sum of all task stacks, whereas event-based systems only need to allocate the maximum of all task stacks, since event-based tasks are executed sequentially in a non-preemptive manner. SLOTH supports event-based applications in its non-preemptive configuration.

[2] The concept of continuations by Draves et al. [DBRD91] does not save blocked state on a stack, but in a different dedicated memory area instead (see discussion in Section 7.3.1).

[3] In this thesis, higher priority numbers correspond to semantically higher priorities.

Operating system object	Properties
`Task1`	Priority 1, auto-started upon system start-up
`Task3`	Priority 3
`Task4`	Priority 4
`ISR2`	Priority 2, connected to serial interface receive interrupt
`Res1`	Resource accessed by `Task1` and `Task3`
`Res2`	Resource accessed by `Task1` and `Task3`
`Alarm1`	Alarm activates `Task4` upon expiry

Table 4.1: Abstract application configuration for an example event-triggered real-time application. OSEK specifies a dedicated configuration language named OIL (OSEK implementation language) [OSE04]; SLOTH uses a Perl-based domain-specific language to specify the configuration for easy processing (see Section 5.1).

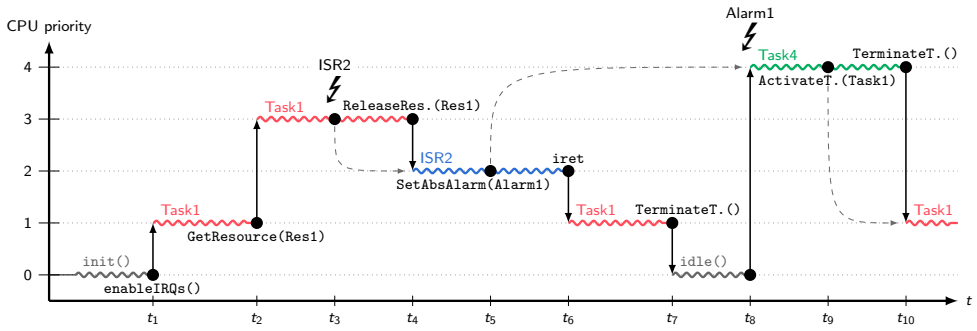


Figure 4.1: Example control flow trace in a SLOTH system. The control flow trace uses the configuration of the application example introduced in Table 4.1.

4.2.1 Basic Tasks

The main idea for the design of the task abstraction in the SLOTH operating system is to assign every application task to a dedicated interrupt source and to have the task code be executed by the attached interrupt service routine. Additionally, all of those interrupt sources are initialized by the operating system to have an interrupt priority that corresponds to the task priority as supplied by the application configuration. Figure 4.2 shows the SLOTH design for the example application specified by the configuration in Table 4.1; the priorities of the interrupt sources assigned to `Task1`, `Task3`, and `Task4` are set to the configured task priorities.

The activation of a task (by using the system call `ActivateTask()`) makes the SLOTH system trigger the interrupt request of the task's interrupt source by setting the interrupt request bit from within the operating system software. This way, the arbitration unit of the interrupt controller, which all interrupt sources are connected to, automatically initializes a new arbitration round to figure out which of the interrupt requests that are currently pending has the highest priority.

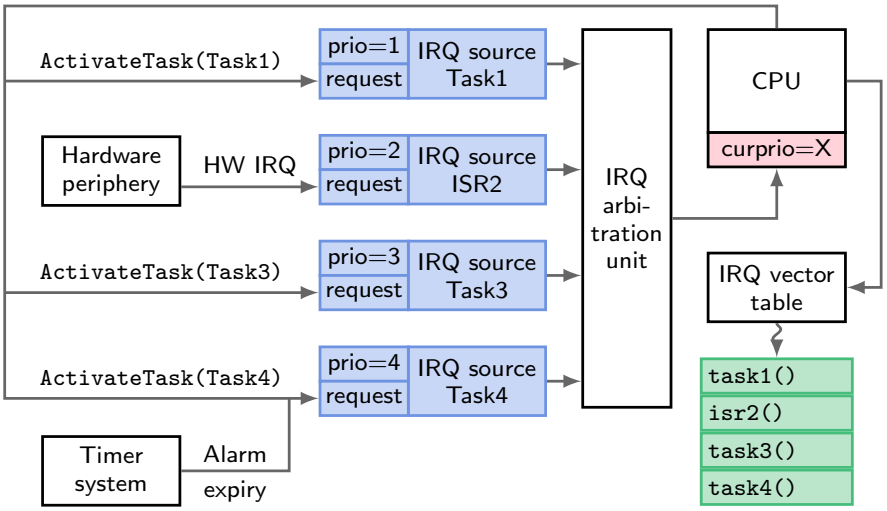


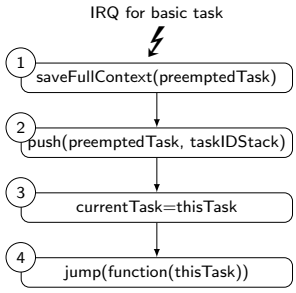
Figure 4.2: Design of the SLOTH operating system. The design targets a system with stack-based basic task execution and shows a SLOTH system tailored to the example application introduced in Table 4.1.

If that priority is higher than the current CPU priority, it will interrupt the CPU execution, look up the corresponding interrupt handler from the interrupt vector table, and dispatch it. Since SLOTH directly attaches the task code functions as supplied by the application to the vector table (see Figure 4.2), the task is immediately dispatched if the priority situation permits. In the example control flow in Figure 4.1, `Task4` activates `Task1` at t_9 by setting its request bit. Since the pending interrupt priority of one is lower than the current CPU priority of four (since `Task4` is currently running), the interrupt controller does not preempt `Task4`, which is the correct semantics in a priority-driven system.

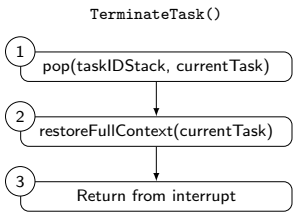
Since the interrupt hardware takes care of the scheduling and dispatching of SLOTH tasks, little has to be done in software. Only a very small task wrapper function, which contains only few machine instructions, is executed before jumping to the user task function (see Figure 4.3(a)). The wrapper only takes care of saving those parts of the register context that the interrupt hardware has not yet saved automatically (Step (1) in Figure 4.3(a))[4]; additionally, it saves the task identifier of the preempted task on a stack ((2)) and sets the current task identifier to the newly dispatched task ((3)). The current task identifier field is needed by the operating system as an index into operating system data structures, as well as for the implementation of the system service `GetTaskID()`, which returns the identifier of the current task.

The opposite action is taken by the SLOTH operating system when a task terminates by issuing the `TerminateTask()` system call (see Figure 4.3(b)). The

[4]Note that, with appropriate compiler support, the wrapper could save only those registers that are actually *used* by the dispatched task.



(a) Prologue for a basic task in SLOTH.



(b) Actions taken by the `TerminateTask()` system call in SLOTH.

Figure 4.3: Steps taken by the SLOTH basic task wrapper. Different steps are taken (a) when a task is dispatched in the SLOTH prologue and (b) when a task terminates by executing `TerminateTask()`.

operating system then restores both the identifier (Step (1) in Figure 4.3(b)) and the context of the preempted task ((2)). After that, it returns from handling the current interrupt (which corresponds to the user task), usually using a special return-from-interrupt instruction from the platform's instruction set ((3)). This triggers the corresponding sequence in the CPU and interrupt hardware to restore the part of the context that has been automatically saved upon interrupt entry, returning to the execution of the preempted task where it left off. The return from interrupt also restores the previous interrupt priority as the current CPU priority, which in SLOTH corresponds to the task priority of the restored task. In the example control flow in Figure 4.1, `Task1` terminates at t_7, which restores the previous interrupt and CPU priority of zero that was active before that task was dispatched—in this case, the system's initialization and idling routine (see Section 4.2.7). When `Task4` terminates at t_{10}, the return from interrupt also restores the idle routine's interrupt priority of zero, but in that case, the activation of `Task1` is still pending at priority level one. Thus, `Task1` is dispatched before really returning to the idling routine.

The system service `ChainTask()` is special in the sense that it combines the activation of a successor task with the termination of the current task. In a preemptive system, however, care has to be taken that a higher-priority successor task is not dispatched until the current task is fully terminated—meaning that no state of that task is still on the stack. This is especially important in the situation when a task chains itself, which would otherwise lead to a stack overflow eventually. To reach the desired behavior, the SLOTH operating system first disables interrupts for a short and bounded amount of time before activating the task by setting its pending bit and terminating the current task. This way, the interrupt handler prologue corresponding to the new task is not dispatched un-

til the return-from-interrupt instruction is executed, which implicitly re-enables interrupts.

Note that the presented design targets hardware systems that have as many IRQ sources as there are tasks in the application—the ideal case. The hardware-centric SLOTH design generally also allows to multiplex IRQ sources in software, traded for a reduction in the improvement of the non-functional system properties. Other multiplexing mechanisms for SLOTH implementations on less capable hardware platforms include the timer cell multiplexing for task activations and deadline monitoring in time-triggered systems, which is discussed in Section 4.4.8.

4.2.2 Interrupt Service Routines

Control flows that are *asynchronously* activated by hardware periphery are encapsulated in ISRs. Operating systems usually distinguish between ISRs that are allowed to use system services (called category-2 ISRs in SLOTH[5]) from ISRs that must not call system services (called category-1 ISRs[6]). The idea is that applications can optimize those restricted category-1 ISRs to have lower worst-case dispatch latencies because they cannot be delayed by an operating system that is synchronized in a blocking manner (see also Section 2.2.1); this is because they are not allowed to issue system calls, which is the only way to modify operating system state.

In the SLOTH design, category-2 ISRs are designed almost identically to basic tasks, except that their wrapper functions do not alter the current task identifier so that the system call `GetTaskID()` can return the correct value. Since they can access operating system state by issuing system calls, category-2 ISRs need to be respected by the operating system synchronization, which SLOTH performs in a blocking manner by raising the CPU priority to the highest priority of all tasks and category-2 ISRs in its critical sections. This way, their execution is delayed until after the critical section inside the operating system is left; at that point, SLOTH lowers the priority to its previous value. As argued in Section 2.2.1, SLOTH only has few software data structures and system-internal critical sections (see also Section 5.5.5), and it synchronizes them in a blocking way due to its simplicity and its small resource footprint in software, which is one of the main SLOTH design themes and requirements. Additionally, the overhead introduced by wait-free non-blocking algorithms, which would be needed to guarantee real-time properties, can surpass the lengths of the critical sections themselves by a large factor, which would mitigate the advantages of such synchronization—especially if the hardware platform does not offer an atomic read–modify–write instruction.

[5] Other operating systems call those types of ISRs deferred procedure calls (DPCs), bottom-half handlers, or epilogues.

[6] Other operating systems call those types of ISRs top-half handlers or prologues.

The same priority alteration mechanism is also used to implement the system services to suspend and resume execution of category-2 interrupts (`SuspendOSInterrupts()` and `ResumeOSInterrupts()`) so that the application itself can synchronize (in a blocking manner). Since SLOTH runs both tasks and ISRs as interrupt handlers with interrupt priorities, ISRs are scheduled and dispatched in the same priority space as tasks, yielding a unified priority space. In the example control flow in Figure 4.1, for instance, the interrupt at t_3 is delayed until after `Task1` lowers its priority at t_4 by releasing its resource (see Section 4.2.3). When `ISR2` terminates at t_6, it executes a regular return-from-interrupt instruction and thereby implicitly re-activates the pending control flow with the next-highest priority, `Task1`. Thus, SLOTH applications can distribute their priority slots freely among tasks and ISRs based solely on their intended *semantic* priorities—completely preventing the problem of rate-monotonic priority inversion (see Section 2.2).

Category-1 ISRs in SLOTH are simply assigned higher interrupt priorities than all tasks and category-2 ISRs in the system. This way, they are not delayed by the operating system synchronization; since they are not allowed to issue system calls, they cannot potentially corrupt operating system state. The system calls offered to the application to suspend or disable *all* interrupts (`SuspendAllInterrupts()` and `DisableAllInterrupts()`), which include those of both category 1 and category 2, are performed by disabling interrupts on the platform; resuming or enabling them using `ResumeAllInterrupts()` and `EnableAllInterrupts()` is performed by re-enabling interrupts. An alternative design instead raises the CPU priority to the highest priority of *all* control flows in the system—tasks, category-2 ISRs, and category-1 ISRs—which is crucial for mixed-mode systems with a time-triggered SLOTH ON TIME system running on top (see Section 4.4.6).

4.2.3 Resources

SLOTH applications can synchronize their task control flows using the resource abstraction for mutual exclusion, which is designed using a stack-based priority ceiling protocol similar to Baker's stack resource policy [Bak90] (see related work in Section 7.3.1) to avoid deadlocks and unbounded priority inversion (see also Section 2.2.1). When a task acquires a resource by calling `GetResource()`, its priority is temporarily raised to the highest priority of all tasks that can potentially acquire that resource—the resource's ceiling priority. This way, tasks can never become blocked upon resource acquisition and the acquisition will always succeed; otherwise, another task with a higher priority (gained by acquiring that very same resource) would be running instead. Furthermore, dispatching of other critical tasks is delayed until after the critical section is left by calling `ReleaseResource()`, which is when the original priority is restored.

The SLOTH design incorporates resources by directly altering the CPU priority as is done for the operating system synchronization (see Section 4.2.2). By raising

the CPU priority and lowering it again after releasing the resource, the interrupt controller automatically dispatches any critical tasks that were activated during the execution of the critical section. Since multiple resources can be acquired—although only in a strictly nested manner—the previous priority has to be saved on a stack-like data structure in the operating system. Because of the static system configuration, the stack usage induced by resource acquisition can be statically bounded at compile time.

In the example control flow in Figure 4.1, `Task1` acquires a resource it shares with `Task3` (see application configuration in Table 4.1) at t_2, so SLOTH raises the current CPU priority to the resource ceiling priority of three. `ISR2`, which is triggered during the critical section at priority two, is delayed until after `Task1` leaves the critical section by releasing the resource at t_4, trying to restore its original priority of one.

Since SLOTH tasks and category-2 ISRs are basically designed the same way, category-2 ISRs can seamlessly participate in the priority ceiling protocol using resources. Thus, SLOTH applications can use the resource abstraction not only to synchronize among tasks, but also between tasks and ISRs. The OSEK specification, for instance, classifies such a feature as being *optional* [OSE05, p. 32], but it is provided by the SLOTH design without additional design nor implementation effort.

4.2.4 Alarms

Alarms are the timer abstraction of the event-triggered SLOTH system; they can be configured by the application to activate a task after a specified amount of time has elapsed. If a task is statically configured to be possibly activated by an alarm at run time, SLOTH assigns that task to an interrupt source that is connected to the platform's timer system (see `Task4` in Figure 4.2). This way, after the timer is correctly configured during the initialization and by the system calls `SetAbsAlarm()` and `SetRelAlarm()`, which set an alarm expiry to be scheduled at run time, that task is automatically scheduled and dispatched by the timer and interrupt hardware if the currently running task has a lower priority, since the timer expiry leads to the timer system requesting the appropriate interrupt. In the example control flow in Figure 4.1, `ISR2` sets an alarm at t_5 to expire at t_8, configured to activate `Task4` (see Table 4.1). At that point, the hardware timer subsystem sets the interrupt request for `Task4`'s interrupt source, which is configured at priority four (see Figure 4.2). Thus, `Task4` is scheduled and immediately dispatched at t_8—concurrently and automatically by the platform's timer and interrupt subsystems.

The OSEK OS specification additionally allows applications to specify callback functions to be called upon alarm expiry instead of activating a task. OSEK callbacks are not allowed to use system services and were introduced to provide a low-overhead means to take action after a specified time budget has elapsed. However, since SLOTH tasks are already very light-weight since their wrapper

functions are reduced to the necessary instructions, providing this additional abstraction would not have an extra benefit for SLOTH applications. Thus, an OSEK OS interface for callback functions in SLOTH can be transparently mapped to high-priority basic tasks in the SLOTH configuration.

4.2.5 Non-Preemptive Systems, Scheduler Locks, and Internal Resources

The basic SLOTH design as described so far targets a *preemptive* system: Activations of higher-priority tasks by lower-priority tasks lead to an immediate scheduling decision entailing preemption and dispatching of the new task. In SLOTH systems, such preemptions are taken out automatically by the interrupt controller.

SLOTH can also be configured by the application to act in a *non-preemptive* manner, which still makes the operating system schedule the application tasks according to their configured priorities, but which does not preempt tasks once they have been dispatched—even if a higher-priority task has become ready to execute in the meantime. Such non-preemptive task execution is designed in SLOTH by enhancing the small task wrapper that is executed at the beginning of all tasks to set the CPU priority to the maximum priority of all tasks in the system. All tasks will therefore execute at the same priority—the highest task priority—at run time. This way, activations of high-priority tasks do not lead to a preemption; however, if several tasks are ready at the same time, the one with the highest priority will be scheduled next by the interrupt hardware. This happens when a running task terminates or chains another task for execution after termination, which is the desired semantics in such a non-preemptive event-triggered system.

Additionally, a non-preemptive system implements a system call `Schedule()`, which states an explicit point of re-scheduling for tasks to invoke, effectively setting an explicit preemption point in the whole real-time system. In SLOTH, that system call first lowers the CPU priority to the original task priority before re-raising it to the maximum task priority. This way, the interrupt controller will dispatch any tasks with a higher priority that are ready to run. Once they have completed, the original task will be continued in its execution. In preemptive systems, `Schedule()` has an empty implementation since re-scheduling is performed immediately once the task ready list (corresponding to pending interrupts in SLOTH) is altered.

In a manner that is similar to non-preemptive systems, a scheduler lock can be designed, which a task in a preemptive system can acquire during a critical section to delay *all* tasks in the system. This scheduler lock can be regarded as a special virtual resource `RES_SCHEDULER`, whose ceiling priority is set to the highest of all tasks in the system. This resource can be transparently used by tasks in SLOTH to implement non-preemptive sections using the regular resource system calls as described in Section 4.2.3.

Additionally, groups of tasks that the application has configured not to preempt each other can be implemented in a related way. SLOTH offers *internal* resources for that purpose, which are special resources whose ceiling priority is set to the highest priority of all tasks in that group. Internal resources are automatically acquired once the task starts running, and they are automatically released when the task terminates, resulting in the desired semantics of preemption groups. SLOTH designs internal resources analogously to the way non-preemptive systems are designed: The corresponding task wrappers include an instruction to raise the priority to the ceiling priority of the internal resource when the task is dispatched. Explicit re-scheduling via `Schedule()` works the same way as in completely non-preemptive systems.

Non-preemptive tasks, groups of tasks, or completely non-preemptive systems have two main advantages. For one, the participating tasks are implicitly synchronized with each other, since they are executed atomically to each other. Second, non-preemptive tasks can be used to significantly reduce the stack requirements of a real-time application, thus saving costly RAM [DMT00]. This is because only the maximum stack usage of the participating tasks has to be allocated and not the sum of all stack usages as in preemptive systems. On the other hand, non-preemptive tasks bear a potentially high release jitter, since their activation can be delayed by other participating tasks.

4.2.6 Multiple Task Activations

An optional feature related to task activations is the ability of the operating system to record *multiple*, queued activations of the same application task. In SLOTH, this feature is implemented in software using a software counter, since interrupt controllers do not have the ability to store more than one pending interrupt request per source. Thus, if the multiple activation feature is enabled in the system configuration as provided by the application developer, the activation of a task first increments the task's software activation counter before setting its IRQ pending bit as in the standard implementation. The activation counter is decremented in the trailing wrapper of the task that is about to terminate; if the counter is still greater than zero, then the IRQ source is triggered again before the task is really terminated. This way, the task is scheduled again if it still has the highest priority in the system at that point.

Since it is implemented in software, this mechanism only works for tasks that are only activated through the software system call interface. It does not work for ISRs and task activations that are triggered by alarms, since their pending bits are set by the periphery hardware and timer hardware, respectively; thus, SLOTH cannot add software instructions to increment a counter. Hence, the hardware-centric SLOTH design can only make use of hardware features as far as possible with a given hardware platform.

4.2.7 System Start-Up and Idling

The SLOTH system start-up—like the rest of the system—is focused on the interrupt subsystem. The platform start-up code usually runs with disabled interrupts and initializes the main stack, which all SLOTH tasks and interrupt handlers are executed on, the interrupt vectors, and platform-specific registers. After that, SLOTH initializes the interrupt sources configured to be used for SLOTH tasks; this step entails setting the source interrupt priorities to the configured task priorities and already setting the request bits for those tasks that are configured to be auto-started upon start-up by the application developer. The scheduler is then started by simply enabling interrupt recognition; since the start-up interrupt priority is zero, any auto-started task will preempt the system at that point and start executing. In the example control flow in Figure 4.1, the initialization runs at priority level zero with interrupts disabled; after that, the scheduler is started by enabling interrupts at t_1. Since `Task1` is configured by the application to be auto-started (see Table 4.1), its request bit was set during the initialization, which is why it is immediately dispatched at t_1.

Since the SLOTH start-up runs at priority zero, the point after the enable-interrupts instruction will be the point where the system returns to when it is idle and all tasks have run to completion (see t_7 in the example control flow in Figure 4.1). Thus, the system's idle loop is executed at that point and can put the parts of the hardware platform to sleep or to a low-power mode until the next interrupt occurs—meaning that a SLOTH task has become ready to execute (see t_8 in Figure 4.1).

4.2.8 SLOTH Summary

By mapping application tasks to hardware interrupt service routines and interrupt sources, SLOTH can implement a fully-featured event-triggered real-time operating system, as exemplified by the OSEK OS standard in its BCC1 conformance class, for instance. The management of basic tasks, application ISRs, resources for synchronization, and alarms is implemented by utilizing the interrupt system and its interrupt priorities, making SLOTH systems simple and efficient. After an interrupt request has been triggered, its source—a periphery device or the CPU itself—and the requested type of control flow—a task, ISR, or callback—is completely oblivious to the CPU; it automatically dispatches the corresponding control flow if the current CPU priority is below the requested priority. This unification of control flows and priority spaces completely prevents rate-monotonic priority inversion (see also evaluation in Section 6.5). Additionally, advanced operating system features like preemption groups and multiple activations, which are demanded by some real-time applications, can be added by slightly adapting and enhancing the SLOTH abstractions and mechanisms. Table 4.2 shows a summary of the SLOTH design of embedded operating system features.

Operating system feature	SLOTH design
Basic task activation	Set request bit of corresponding interrupt source.
Basic task termination	Execute return-from-interrupt instruction.
Basic task chaining	Disable interrupts, set request bit, and return from interrupt.
Basic task execution	Task wrapper saving context and preempted task ID, setting new task ID.
Category-2 ISR execution	ISR wrapper saving context.
Category-2 ISR suspension	Set CPU priority to highest of all tasks and category-2 ISRs.
Category-1 ISR execution	ISR wrapper saving context.
Category-1 ISR suspension	Disable interrupts.
Resource acquisition	Raise CPU priority to resource ceiling priority.
Resource release	Lower CPU priority to previous priority.
Alarm-activated tasks	Use interrupt source connected to timer system for that task.
Non-preemptive system	Set CPU priority to highest of all tasks in the task wrappers.
Scheduler lock	Special resource with ceiling priority set to highest of all tasks.
Internal resources	Set CPU priority to highest of all tasks in the same group in the task wrappers.
Multiple task activations	Software counter incremented upon activation and decremented in trailing task wrapper.
Auto-started tasks	Request bits set by start-up routine.

Table 4.2: Summary of the way that the SLOTH design implements embedded operating system features. SLOTH implements services related to task execution, ISR execution, coordination and synchronization, and it implements advanced real-time services.

4.3 SLEEPY SLOTH Systems with Blocking Task Execution

The SLOTH system design presented so far targets systems with strictly stack-based task execution patterns, supporting applications that feature only basic tasks, which run to completion. Such tasks can be ideally mapped to be executed as interrupt service routines on interrupt controllers with priority-based preemption following the SLOTH approach. Although the presented feature set supports a wide range of embedded real-time applications, some applications additionally need *task blocking* functionality for flexibility in the specification of complex tasks. In SLOTH, tasks that can potentially block at run time are called *extended tasks*.[7] Such systems exhibit interleaving task execution, where high-priority task execution can be interleaved by low-priority task execution during

[7] General operating system literature also calls such systems to be process-based.

the time that the high-priority task is blocked. From the implementation point of view, extended tasks cannot save their state on a common stack when blocking—as do basic tasks when being preempted. Instead, when an extended task has to block, it either has to explicitly save its context to a dedicated memory area (as known from the Mach continuation model [DBRD91]), or it uses a run time stack of its own, which is simply switched upon blocking. Although the Mach continuation model would reduce the application's data memory requirements, it would also increase the execution latency by having to save selected data upon blocking, and, more importantly, it would require the application to review each potential blocking point and state which parts of its run time context it will need after being unblocked (see also discussion of related work on execution semantics in Section 7.3.1). In order to allow the application programmer for easy and transparent programming of his tasks, SLOTH opts for the dedicated-stack model. Thus, the SLEEPY SLOTH operating system—an extended version of the SLOTH operating system—has to switch the extended task stacks at appropriate points in the execution.

An application developer might prefer an implementation of parts of his application in an extended task over an implementation in a basic task for reasons of flexibility and application code simplicity. Consider a computation that is divided in two parts, the second of which depends on the occurrence of an independent event, such as the availability of an additional piece of data. In an extended task system, this computation can be implemented in a single task that simply executes a blocking system call to wait for that event, which makes the operating system save the complete task context including its stack. After the event has occurred, the extended task is unblocked by the operating system and can transparently continue to execute on its data and stack. This scenario can also be implemented using two basic tasks, the first of which terminates when it semantically "blocks", and the second of which is activated by the operating system when the event occurs, corresponding to "unblocking" it. In that case, however, the application developer manually has to consider which state to save and restore in order to explicitly transfer state between the two tasks—as in the Mach continuation model discussed above. This process is termed stack ripping and is error-prone especially when evolving code, and it exacerbates with multiple blocking calls and language loops involved [AHTBD02; KKK07]. Additionally, such a manual split into several subtasks is not feasible for blocking code in libraries, which would need to distinguish between different task callers. With extended tasks, the operating system takes care of saving the whole register set and stack—automatically and transparently for the application developer.

Table 4.3 shows the configuration of an example application that features two extended tasks `ET3` (with an associated event `Ev1`) and `ET5` and two basic tasks `BT1` and `BT2`, and one resource `Res1`. Figure 4.4 shows an example control of an interleaving task execution in that application; the details are explained while describing SLEEPY SLOTH's design in this section. The execution of the high-priority extended task `ET3` is interleaved by the execution of the low-priority tasks

Operating system object	Properties
`BT1`	Basic task, priority 1, shared basic task stack, auto-started upon system start-up
`BT2`	Basic task, priority 2, shared basic task stack `stk_bt`
`ET3`	Extended task, priority 3, stack `stk_et3`
`ET5`	Extended task, priority 5, stack `stk_et5`
`Ev1`	Event assigned to `ET3`
`Res1`	Resource accessed by `BT1` and `ET3`

Table 4.3: Abstraction application configuration for an example event-triggered SLEEPY SLOTH application with extended tasks. The application's basic tasks use the prefix `BT`, whereas its extended tasks use the prefix `ET`.

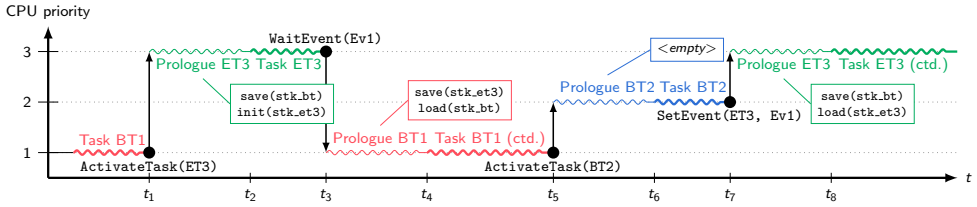


Figure 4.4: Example control flow trace in a SLEEPY SLOTH system with extended tasks. The trace shows a system with extended tasks and uses the application example introduced in Table 4.3. The figure does not reflect timing proportions, as the SLEEPY SLOTH task prologues are usually very short compared to the task functions themselves.

`BT1` and `BT2` while `ET3` is blocked between t_3 and t_7. From an API point of view, the system calls in extended task handling are `WaitEvent()` and `SetEvent()`. The former blocks if the associated current event mask of the current task does not include the event that the task wants to wait for; the latter sets the specified event in the event mask of the specified task, unblocking it if it has been waiting for it.

The SLEEPY SLOTH operating system aims to support extra flexibility through extended tasks in addition to basic tasks while preserving the SLOTH idea of letting the interrupt controller do the scheduling and dispatching for both kinds of tasks if the hardware permits—in order to preserve its simplicity and low-overhead advantages as well as its unified priority space to prevent rate-monotonic priority inversion.

The challenges in the SLEEPY SLOTH design, which combines the two worlds of a blocking control flow abstraction—the extended task abstraction in SLEEPY SLOTH—and an efficient, interrupt-driven operating system, are threefold. The first and main challenge is the ability to suspend task execution and resume its execution later, which interrupt controllers do not support for interrupt handler

execution. This is due to the fact that interrupt handlers are supposed to run to completion *transparently* to the interrupted control flow—this semantics is hard-coded and guaranteed by the interrupt hardware. Thus, for extended tasks, SLEEPY SLOTH needs to find a way to implement both the *suspension* of a blocked ISR and the *re-activation* of an unblocked ISR, saving and restoring its full context including its dedicated stack appropriately—effectively bringing the software execution model as known from the thread abstraction to the ISR abstraction as known by the hardware.

Second, by nature, interrupts are *asynchronous* in their occurrence; that is, no prediction can be made as to where exactly a control flow yields the CPU when interrupted. Thus, performing the necessary context and stack switch in the *preempted* extended control flow *before* dispatching is impossible in a SLOTH-like system with hardware-triggered preemptions, since the interrupt scheduler and dispatcher are provided by the hardware.

The third challenge regards the execution efficiency of the resulting SLEEPY SLOTH system: The added flexibility for the application developer to be able to use extended tasks should not come at the price of lowered system performance. Especially the latencies for scheduling and executing basic run-to-completion tasks, which do not need stack switches when running among other tasks of their class on the same shared stack, should remain comparable to the original SLOTH operating system.

In order to tackle those challenges, the central design idea in SLEEPY SLOTH is the deployment of sophisticated task prologues, which replace the basic task wrappers. The prologue determines what kind of context switch is needed whenever the corresponding task is dispatched and is prepended to the task interrupt handler. It is thus executed both when the task is about to run for the first time *and* when its execution is resumed after being blocked or preempted. Additionally, the interrupt disable bits of the individual interrupt sources assigned to tasks are used to implement blocked tasks utilizing the hardware platform. Figure 4.5 shows an overview of the modified SLEEPY SLOTH design for the example application configured as shown in Table 4.3; compare the design to the original SLOTH design in Figure 4.2.

In effect, SLEEPY SLOTH allows basic tasks to be implemented as run-to-completion ISRs internally, whereas complex control flows can be implemented as extended tasks—also running as hardware ISRs internally, however. Thus, it combines flexible execution semantics—allowing SLEEPY SLOTH tasks to be activated by hardware or by software and to run to completion or to block—with efficient run time properties—since they are scheduled and dispatched by the interrupt hardware.

4.3.1 Extended Task Dispatch via Prologues

In SLEEPY SLOTH, the main difference in extended task execution compared to basic task execution is the need to switch stacks (and, thereby, the full con-

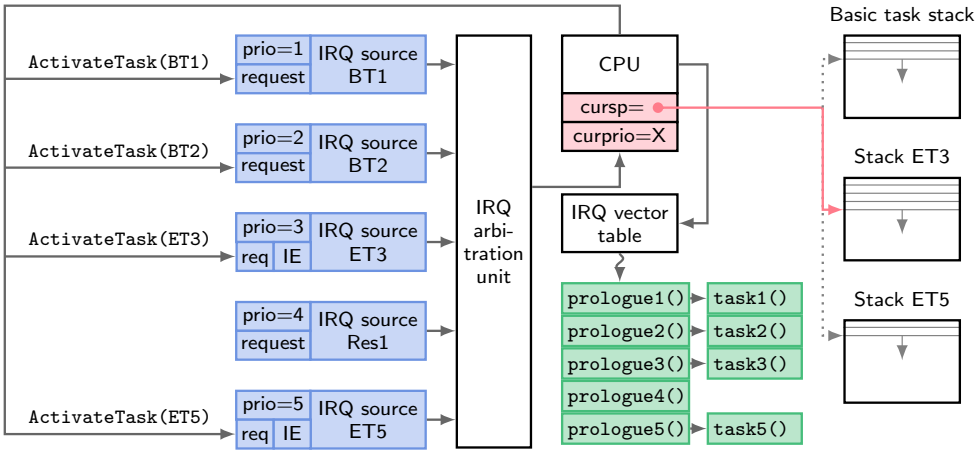


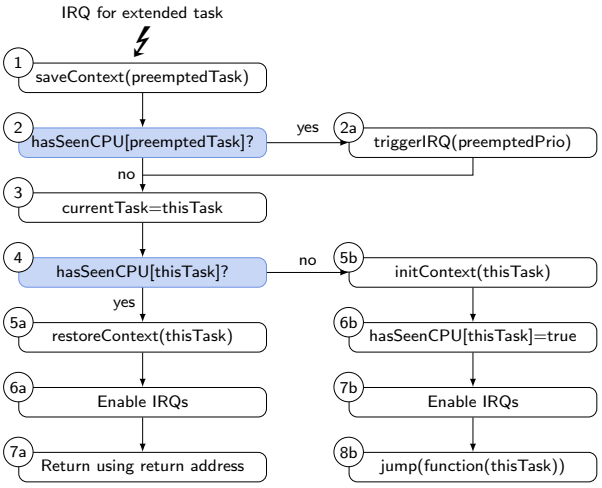
Figure 4.5: Design of the SLEEPY SLOTH operating system. The design shows a SLEEPY SLOTH system tailored to the example application introduced in Table 4.3.

texts) in task switches that involve extended tasks. In traditional systems, the operating system software features a context switch function that can be augmented accordingly. However, since scheduling decisions in SLOTH are performed asynchronously by the interrupt controller on appropriate hardware and not synchronously by operating system software, such a context switch function does not exist in software and can therefore not be instrumented. Instead, the necessary stack switch has to be performed when the next task has already been dispatched by the hardware. The reason for that necessity is that the hardware semantics of an interrupt entails saving the context of the interrupt control flow plus dispatching the new control flow with a *pristine* context; for an unblocked task, however, the new control flow needs to *restore* its context and stack pointer.

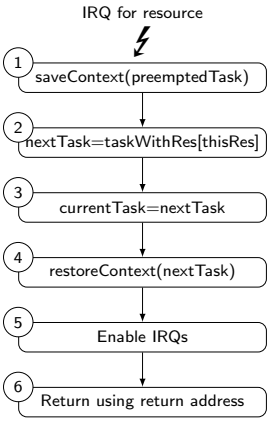
SLEEPY SLOTH therefore performs the stack switch in a *task prologue* in the *successor* control flow; the prologue software is executed in the wrapper before the actual user task function (see the vector table in Figure 4.5). The task prologue is the single point to decide whether to save the stack of the interrupted task and whether to restore or to initialize the stack of the dispatched task. Figure 4.6 shows the steps that are taken by the SLEEPY SLOTH prologues when extended tasks, basic tasks, and preempted resource-holding tasks are dispatched; depending on the situation, they need to figure out what kind of control flow they interrupted in their steps. The following paragraphs discuss the prologue for extended tasks as depicted in Figure 4.6(a); the prologues for basic tasks and resources are discussed in Section 4.3.5 and Section 4.3.6, respectively.

First, the extended task prologue saves the extended context of the interrupted task (i.e., those registers that have not automatically been saved by the hardware when dispatching the interrupt handler[8]) to the corresponding task

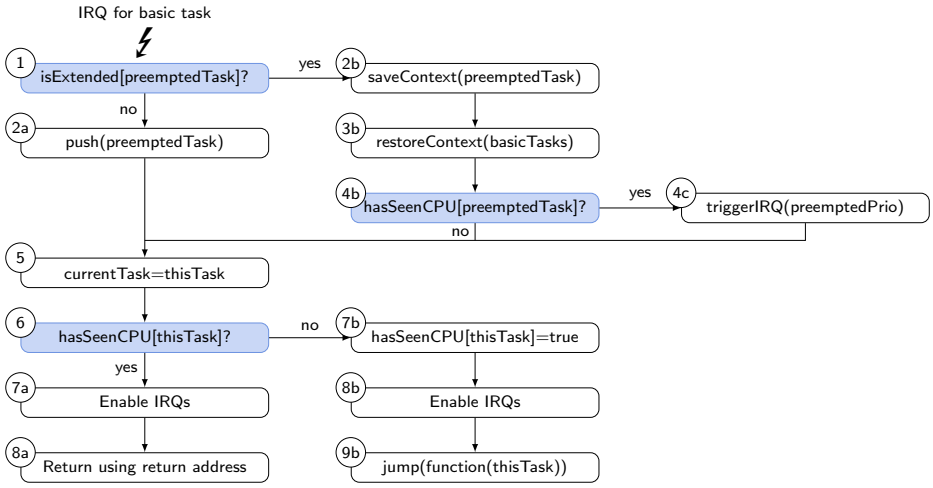
[8]Again, note that, with appropriate compiler support, the prologue could save only those registers that are actually *used* by the dispatched task.



(a) Prologue for an extended task in SLEEPY SLOTH.



(b) Prologue for a resource-holding task in SLEEPY SLOTH.



(c) Prologue for a basic task in SLEEPY SLOTH.

Figure 4.6: Steps taken by the SLEEPY SLOTH task prologues. Different steps are taken when (a) an extended task, (b) a continued part of a task with an acquired resource, and (c) a basic task is dispatched.

stack, whose stack pointer in turn is saved to a context array in the operating system (Step (1) in Figure 4.6(a), and t_1, t_3, and t_7 in Figure 4.4). Next, the prologue checks whether the interrupted task was either preempted or blocked, or whether it terminated ((2)). If it did *not* terminate, the prologue re-activates the task to be continued later by triggering the task's interrupt source at the priority it was running at ((2a)). This step corresponds to putting the preempted

task back into the ready-tasks list in the operating system, which in the SLOTH systems is implemented in the IRQ pending bits (a blocked task is kept from being scheduled by clearing its interrupt enable bit as described in Section 4.3.3). Task `ET3`'s prologue performs this step at t_1 and t_7 in the example control flow in Figure 4.4. Note that the interrupted task's priority might have been raised at the time of preemption due to the possession of a resource in combination with the stack-based priority ceiling protocol; in that case, the unblocked task needs to be treated specially (see Section 4.3.6). The check for that condition itself is done via a bit array maintained by the operating system (`hasSeenCPU[]`). After that, the operating system variable holding the current task is set to the dispatched task ID, which corresponds to the current IRQ number (③).

If the dispatched task has run before (checked by comparing its `hasSeenCPU` property; Step ④ in Figure 4.6(a)), its context is restored from the context array in the operating system (entailing its stack and registers; ⑤a), IRQs are enabled (⑥a), and execution of the task is continued by returning via the return address in the saved context (⑦a). In the example control flow in Figure 4.4, this happens at t_3 and t_7.

If the dispatched task has *not* run before, its context is initialized (entailing resetting its stack pointer; ⑤b) and its `hasSeenCPU` property is set to true to be considered by further preemptions by other tasks (⑥b). Eventually, the task prologue enables IRQs (⑦b) and jumps to the actual user task function to start executing user code (⑧b). In the example control flow in Figure 4.4, this happens when task `ET3` is dispatched for the first time at t_1.

4.3.2 Extended Task Termination

Task termination in SLEEPY SLOTH differs from the way it works in the original SLOTH since a stack switch might be needed after termination—for instance, if another task (with the next-highest priority) was unblocked and needs to be continued in its execution after termination of the current task. Again, SLEEPY SLOTH solely relies on the prologue of the next task to determine if a full context switch is needed or not.

The next task is scheduled and dispatched by the hardware by letting the terminating task set the CPU priority to zero. The interrupt system then determines the next-highest priority task that is ready to run. Before that, the terminating task indicates to the prologue of the following task that it has terminated by resetting its own `hasSeenCPU` flag to false. This way, the interrupt source of the terminating task will not be re-triggered for continued execution (see description in Section 4.3.1 and Steps ② and ②a in Figure 4.6(a)).

Note that SLEEPY SLOTH effectively discards the remaining state of the extended task stack when forcing preemption by lowering the CPU priority to zero. If the same task is activated and dispatched again, its prologue will correctly re-initialize its stack using the original top-of-stack pointer after checking its `hasSeenCPU` flag (see Section 4.3.1).

In contrast to SLOTH systems, SLEEPY SLOTH systems do not return to the start-up routine when idling (compare to Section 4.2.7) due to the technical way of extended task termination. Instead, when the last extended task that is ready to run in a system terminates or blocks, it continues to execute after trying to yield the CPU by setting its priority to zero. Thus, if it was the last ready task in the system, it can jump directly to the system's idling routine, which can put the hardware platform to sleep or to a low-power mode until a new task activation in the form of an interrupt comes in. This design guarantees for low-overhead task dispatching in the system also when returning from the idling state.

4.3.3 Extended Task Blocking

The main additional system call that SLEEPY SLOTH provides over the original SLOTH operating system is `WaitEvent()`. The implementation compares the event mask of the current task to the event mask to be waited for, and, if they do not match, blocks the task. This is done by disabling the task's IRQ source (see the design of extended tasks in Figure 4.5); this way, it will not be considered in the interrupt arbitration, which determines the highest-priority interrupt to be handled, effectively removing it from the "ready list" in hardware. After that, the CPU is yielded by setting the CPU priority to zero (much in the same way that a task terminates; see also Section 4.3.2) and letting pending interrupts (corresponding to tasks that are ready to run) be dispatched. In the example control flow in Figure 4.4, this is what happens when task `ET3` blocks at t_3, which is when the interrupt controller dispatches task `BT1` again.

4.3.4 Extended Task Unblocking

In the SLEEPY SLOTH programming model, tasks are unblocked by setting one of the events that the task has been waiting for. SLEEPY SLOTH's system call `SetEvent()` therefore first checks whether that condition is met, and then it unblocks the task by re-enabling its IRQ source and triggering its IRQ (see also t_7 in the example control flow in Figure 4.4). This makes the interrupt controller consider the task in its priority arbitration mechanism and schedule the task according to the system's priority state. Once the unblocked task is dispatched, its prologue sees that it has run before (see Step (4) in Figure 4.6(a)) and restores its context to continue its execution (Step (5a); see also t_8 in Figure 4.4).

4.3.5 Basic Tasks in SLEEPY SLOTH

Providing the application with the flexibility to use extended tasks should not affect the execution of basic tasks; the overhead for basic tasks should remain as low as in the strictly stack-based SLOTH operating system. SLEEPY SLOTH therefore also lets all basic tasks run on a single stack (see basic task stack in Figure 4.5) as does SLOTH; this makes task switches between basic tasks light-weight and

fast, since the hardware automatically saves and restores part of the register set upon interrupt entry and return, and no additional stack switch is needed. Figure 4.6(c) shows the steps taken by a prologue associated with a basic task in a SLEEPY SLOTH system.

In the right path, executed when the basic task has interrupted an extended task, the prologue has to save the full preempted task context (Step (2b)) and restore the common context of all basic tasks ((3b)) before eventually starting or continuing to execute ((9b) and (8a)). In the example control flow in Figure 4.4, the basic task `BT1` notes at t_3 that it follows an extended task and therefore performs the full context switch by saving task `ET3`'s stack and loading the common basic task stack before resuming execution at t_4.

The left path in Figure 4.6(c), corresponding to a preemption of a basic task by a basic task, does not switch full contexts including the stacks. In the example control flow in Figure 4.4, this is what happens at t_5, when basic task `BT1` activates basic task `BT2`, whose prologue observes that it has interrupted a basic task and therefore directly starts executing the `BT2` task function at t_6 without having to switch stacks.

However, additional overhead is incurred in order to *determine* whether a stack switch is needed—that is, whether either the interrupted or the newly dispatched task or both are extended tasks (see Step (1) in Figure 4.6(c)). This property is configured by the application programmer at compile time and stored in a constant bit field in the operating system for fast access. Apart from that, during times in the application when only basic tasks are scheduled and dispatched, the overhead incurred by the SLEEPY SLOTH operating system is very small and comparable to the one incurred by SLOTH (see also the empty prologue at t_5 in Figure 4.4). Since category-2 ISRs and basic tasks are essentially the same in SLOTH systems (see also Section 4.2.2), category-2 ISRs are also prepended a small prologue in SLEEPY SLOTH configurations in order to switch to the common stack if necessary.

Note that it is not possible to have a basic task run on the stack of the extended task that it preempted. If the preemption took place because the extended task blocked, unblocking that extended task would make it use its stack again. If the basic task were then still active, its part of the stack would get corrupted.

4.3.6 Resources in SLEEPY SLOTH

In SLEEPY SLOTH, the resource abstraction, which is used by applications to synchronize critical sections by raising their priorities according to a stack-based priority ceiling protocol, requires special handling due to the necessity to switch stacks upon switching tasks. Consider a task that, having acquired a resource, is preempted by a higher-priority extended task, which therefore performs a stack and context switch (see example control flow in Figure 4.7). When the preempted task continues execution, it has to do so *at the raised priority* of the resource (t_4 in Figure 4.7). In the original SLOTH operating system, a resource's

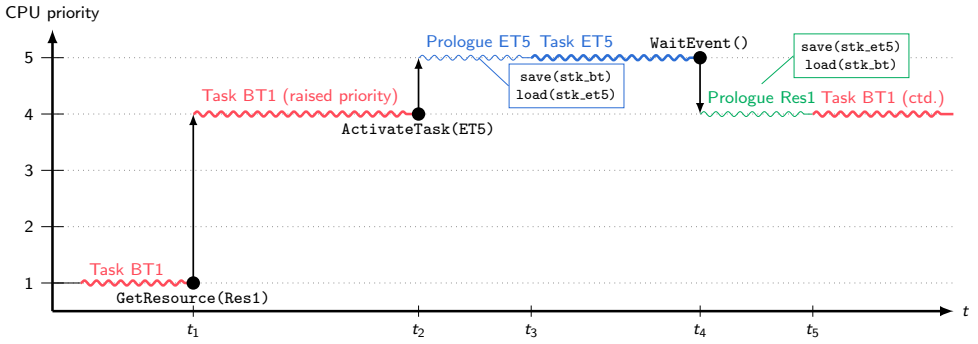


Figure 4.7: Example control flow trace in a SLEEPY SLOTH system with a resource. The trace situation incorporates one basic task `BT1` (with priority 1) and one extended task `ET5` (with priority 5). `BT1` accesses a resource it shares with extended task `ET3` (with priority 3); in SLEEPY SLOTH, the resource is therefore assigned the dedicated priority slot 4 due to the stack-based priority ceiling protocol.

ceiling priority is set to the highest priority of all tasks that can acquire that resource. In order for SLEEPY SLOTH to be able to distinguish between an activation of that highest-priority task and a re-activation of a task that acquired that resource, the resource is given its own dedicated priority, one higher than the ceiling. The resource is therefore also assigned a dedicated IRQ source with a prologue of its own (see the design in Figure 4.5). In the example control flow in Figure 4.7, when task `BT1` acquires `Res1` at t_1, its priority is therefore raised to the dedicated ceiling priority of 4.

The resource's IRQ source differs from task IRQ sources in that it is only triggered after a task that acquired the resource is preempted by a higher-priority task (t_2 in Figure 4.7). The preempting task's prologue will re-trigger the resource IRQ (see also Step (2a) in Figure 4.6(a)). This ensures that the dedicated resource prologue will be dispatched once the CPU priority is lowered again—when the higher-priority task terminates or blocks (t_4 in Figure 4.7). The resource IRQ handler then loads a reference to the preempted task (recorded by the original `GetResource()` system call; see Step (2) in Figure 4.6(b)) and restores its context ((4)). The execution priority is left unchanged, since it is already at the resource priority (t_5 in Figure 4.7). This way, the preempted task continues its execution at the correct priority—the raised resource ceiling priority.

Note that, as in the original SLOTH resource implementation, SLEEPY SLOTH also stores the previous priorities on a stack when acquiring a resource (see also Section 4.2.3). This behavior also leads to correct system semantics in a system with extended tasks, since the acquisition of resources is still strictly stack-based. SLOTH, as well as the OSEK OS specification, demands that resource acquisitions be strictly nested and that extended tasks must not block while in a critical section; they can only be preempted.

4.3.7 Synchronization with the Priority Inheritance Protocol

In its basic-task-only variant, SLOTH provides applications with the ability to synchronize accesses in critical sections using a stack-based priority ceiling protocol using the resource abstraction, implemented in SLOTH and SLEEPY SLOTH as described in Section 4.2.3 and Section 4.3.6, respectively. With the blocking ability of extended tasks, SLEEPY SLOTH can offer an alternative synchronization abstraction to the application: PIP resources, which adhere to the priority inheritance protocol (PIP) [SRL90; Liu00]. The priority inheritance protocol is an efficient means to prevent situations of unbounded priority inversion (see also Section 2.2.1), but in contrast to the stack-based priority ceiling protocol, it does not prevent deadlock situations by design, so only appropriate application design can ensure that deadlocks do not occur at run time. Additionally, due to its behavior in nested critical section, it infers *transitive* blocking with a higher worst-case blocking time.[9] However, this synchronization protocol has the advantage that it does not have a broadband effect as ceiling priorities do, which keep all tasks with a priority between the original task priority and the ceiling priority from running. Priority inheritance only keeps other tasks from running once a high-priority task is executing and wants to access the critical section, passing on its priority to the low-priority task. In SLEEPY SLOTH, the application developer can make that trade-off decision by selecting the kind of SLEEPY SLOTH resource to use.

Concerning the design, in contrast to the stack-based priority ceiling protocol, priority inheritance does not alter the task priority directly when a task enters a critical section. Only when a higher-priority task tries to enter the same critical section does the occupying task inherit the high priority. This way, it can leave the critical section as soon as possible so that the high-priority can continue executing.

SLEEPY SLOTH designs PIP resources by maintaining a software lock and a waiting task pointer. When a task acquires a PIP resource that is not yet locked, it simply acquires the software lock. If it is already locked, SLEEPY SLOTH saves the ID of the current high-priority task and switches contexts to the lock owner, which thereby continues its execution at the inherited priority level of the current high-priority task. When the lock owner then releases the PIP resource, the context is switched back to the original high-priority task, enabling it to enter the critical section. If no task has been waiting to enter the critical section when a task releases a PIP resource, the software lock is simply reset. Since these operations require full context switches including stack switches, PIP resources are only available in the SLEEPY SLOTH operating system variant.

[9] In this context, the term blocking time is not related to blocking tasks, but describes the time that a high-priority task is kept from executing by lower-priority tasks when attempting to enter a critical section.

Operating system feature	SLEEPY SLOTH design
Task dispatching	Task prologue determines whether stack switch is needed or not and whether context needs to be restored or initialized.
Extended task termination	Reset `hasSeenCPU` flag to false and set CPU priority to zero.
Extended task blocking	Set `hasSeenCPU` flag to true, disable IRQ source, and set CPU priority to zero.
Extended task unblocking	Re-enable and trigger IRQ source.
Resources with extended tasks	Use dedicated IRQ source and priority slot, resuming preempted task at raised priority.
Resources with priority inheritance	Switch full context to the lock owner and back, preserving the inherited priority.

Table 4.4: Summary of the way that the SLEEPY SLOTH design implements embedded operating system features. In addition to SLOTH, SLEEPY SLOTH implements services for extended tasks and for priority inheritance support. See also the SLOTH design overview in Table 4.2.

4.3.8 SLEEPY SLOTH Summary

SLEEPY SLOTH implements extended tasks by adopting the SLOTH design, having the operating system run all tasks as interrupt handlers with interrupt priorities if the hardware platform permits. It extends the universal control flow abstraction to be configurable in its execution semantics, set to run-to-completion or blocking by the application developer. Since task switches involving extended tasks need stack switching but basic task switches do not, SLEEPY SLOTH prepends an alternative task prologue to every task function. This task prologue performs the appropriate context initialization or restore operation depending on the involved tasks and their states. Task blocking and unblocking is performed by disabling and re-enabling the task IRQ source to exclude and re-include it in the interrupt scheduling, respectively. To correctly resume preempted tasks with acquired resources, resources are assigned a dedicated IRQ source, priority, and prologue. Resources using priority inheritance are implemented using full context switches. Table 4.4 shows a summary of the way that the SLEEPY SLOTH design implements operating system features for systems with blocking tasks and, thus, interleaving task execution.

4.4 SLOTH ON TIME Systems with Time-Triggered Abstractions

SLOTH and SLEEPY SLOTH both target *event-triggered* real-time systems, which execute tasks and ISRs with their configured priorities in response to system-internal

or external events. *Time-triggered* real-time systems, on the other hand, work by executing a statically defined dispatcher table, which specifies application tasks to be dispatched at certain points in time per hyperperiod. In a very deterministic form, time-triggered systems dispatch tasks in equal-length *frames*, upon whose boundaries the operating system checks whether all tasks scheduled for the current frame have been released and whether tasks from the previous frame have overrun. The SLOTH ON TIME design for such frame-based systems is shown in Section 4.4.1.

The frame-based model bears the disadvantage that is very restricted in finding an appropriate schedule for a given set of tasks; it often needs to slice a task into several parts to make it fit into frames, which increases the overhead for saving and restoring the context between parts. In a more general form, the time-triggered real-time system does not dispatch at frame boundaries, but at arbitrary dispatch points within one dispatcher round, which corresponds to the hyperperiod of frame-based systems. Such an execution model is also exemplified by the automotive OSEKtime OS specification, for instance [OSE01]. The SLOTH ON TIME design for such generalized systems is described starting with Section 4.4.2.

This section uses the term *time-triggered operating system* to define an operating system that supports time-triggered task execution. However, a time-triggered operating system, such as the SLOTH ON TIME system described in this section, can also react on events, which it schedules in aperiodic or sporadic tasks, the latter of which bear hard deadlines and need online acceptance tests.

4.4.1 Frame-Based Time-Triggered Task Execution

If the application developer specifies his time-triggered application using frames, SLOTH ON TIME acts as a cyclic executive and can use a single hardware timer to act as its *frame timer*, which it programs to fire once per frame length. Additionally, it translates the frame specifications within one hyperperiod of the application into an array data structure within the operating system. The interrupt handler for the timer as provided by SLOTH ON TIME then uses a simple look-up algorithm to find the tasks that it needs to dispatch in the current frame, and it dispatches them in a sequential manner. Furthermore, it maintains a pointer to the next task to be dispatched in the current or the next frame.

If the application developer configures SLOTH ON TIME to supply additional monitoring functionality, the timer handler is enhanced in two ways. For one, when it is executed at the beginning of the frame, the handler checks whether it has correctly interrupted the SLOTH ON TIME idling routine or any background task (see also Section 4.4.6). If it has interrupted a periodic task, it has detected a frame overrun caused by any of the tasks scheduled in the previous frame; the handler immediately forwards this exceptional situation to the application by executing a callback function provided by the application developer. Second, the SLOTH ON TIME timer handler checks whether all tasks in the current frame

have been released for execution; otherwise, it also raises an exception to the application.

In the background—that is, when all periodic tasks within one frame have been executed and there is still time until the next frame boundary—SLOTH ON TIME can schedule aperiodic tasks, which are released due to an external event. The operating system therefore maintains a queue of released aperiodic tasks, which it sorts according to an application-defined strategy, such as earliest deadline first, for instance. Whenever the timer handler has executed all scheduled periodic tasks within a frame, it then executes the next aperiodic task until the next frame interrupt fires. It then needs to save the complete context of the interrupted aperiodic task so that it can be resumed during the next background slot. For sporadic tasks, which bear hard deadlines, the system performs an application-provided acceptance test to determine if it can schedule the new sporadic task among the periodic task mix and already accepted sporadic tasks. This way, sporadic events that could not be executed without jeopardizing the execution of the rest of the system are raised as an exception to the application as early as possible for it to be able to take action.

Since periodic tasks or slices thereof do not improve their quality of service by terminating *before* the end of a frame, the scheduled background time within a frame can also be consumed earlier in that frame. This mechanism is called slack stealing, and it improves the average response times of aperiodic tasks in the system [Liu00]. If the hardware platform offers at least a second timer, SLOTH ON TIME configures that dedicated *slack timer* with the statically calculated slack budget at the beginning of a frame and starts executing aperiodic tasks from its queue. If the slack budget of the current frame is exhausted, the slack timer will fire and interrupt any aperiodic task in order to start executing the periodic tasks scheduled for that frame. If the hardware platform does not have a second timer, the hardware-centric SLOTH approach multiplexes the main timer to be both its frame timer and its slack timer by re-configuring the counter in software appropriately.

4.4.2 Generalized Time-Triggered Task Execution

SLOTH ON TIME can also be configured for time-triggered execution with arbitrary dispatch points within one dispatcher round. Table 4.5 shows a configuration of an example time-triggered application that features two tasks executed periodically within a dispatcher round of length 1,000. Figure 4.8 shows an example control flow of a dispatcher round of the example application. Note that in the assumed model, time-triggered tasks do not have an associated priority as do event-triggered tasks, but they do preempt currently running tasks in a stack-based manner (see preemption of the first execution of `TTTask1` by `TTTask2` in Figure 4.8). This preemptive model, inspired by the OSEKtime specification, allows for long-running tasks to be executed as is—without the need to split them up into chunks by the application developer.

Operating system object	Properties
Dispatcher round	Cyclic execution, round length 1,000
`TTTask1`	Time-triggered task, dispatched at time offsets 100 and 600 from round start Deadlines at time offsets 450 and 950 from round start
`TTTask2`	Time-triggered task, dispatched at time offset 200 from round start Deadline at time offset 350 from round start

Table 4.5: Abstract application configuration for an example generalized time-triggered application. A dispatcher round is configured to cyclically dispatch two tasks, one of them twice per round, and to monitor the task deadlines.

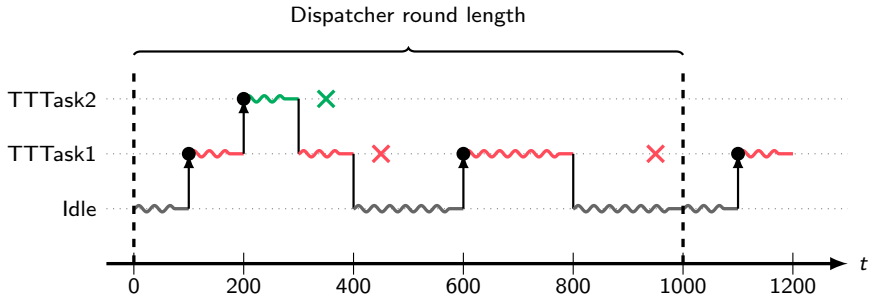


Figure 4.8: Example control flow trace of a SLOTH ON TIME dispatcher round. The trace shows the example application introduced in Table 4.5. Task activations are depicted by circles, their specified deadlines by crosses.

Operating systems implementing this model usually consist of an encoded dispatcher table held in memory and one hardware timer used as a system timer. When the system timer expires, the operating system first re-configures it to the next expiry point in the dispatcher table and then executes the task associated with the current expiry point. Since many current embedded hardware platforms do not only exhibit sophisticated interrupt controllers, which are used by SLOTH and SLEEPY SLOTH, but also sophisticated timer arrays, the time-triggered SLOTH ON TIME variant aims at utilizing those arrays to reduce the memory and latency footprint induced by a time-triggered operating system—if the hardware platform permits.

The main idea in SLOTH ON TIME is to map time-triggered task activations and operating system timing services to individual timer cells of a hardware timer array. By pre-configuring the array according to the static schedule once at system *initialization time*, the overhead incurred by the operating system at *run time* in its productive phase can be kept very low. Additionally, a SLOTH ON TIME system reaches the perfect precision of the underlying timer system. Other operating systems that want to avoid the timer re-configuration cost use a tick-based

Timer cell role	Description
Activation cell	Activates and dispatches the task connected to the associated interrupt source by triggering an interrupt request (see Section 4.4.3).
Control cell	In hierarchically connected timer cells, a top-level control cells enables all lower-order timer cells in order to simultaneously start a whole dispatcher round or schedule table (see Section 4.4.3).
Deadline cell	Triggers the execution of a deadline exception handler when the corresponding point of time in the dispatcher round is reached; the deadline cell is disabled when the corresponding task terminates in time (see Section 4.4.4).
Sync cell	Executes a synchronization routine that modifies the activation and deadline cells to accommodate the detected clock drift (see Section 4.4.5).
Aperiodic cell	Executes a short handler that enables the corresponding aperiodic interrupt source (see Section 4.4.6).
Budget cell	Triggers the execution of a protection hook handler when a task exceeds its execution budget; the budget cell uses a countdown timer that is enabled and disabled when the corresponding task is dispatched and preempted or terminated, respectively (see Section 4.4.7).

Table 4.6: The specific roles that SLOTH ON TIME assigns to individual timer cells. SLOTH ON TIME uses the cells from an available timer array for a distinct purpose each.

timer instead[10], which, however, bears an unavoidable loss in scheduling precision [FGS11] (see also related work in Section 7.3).

Depending on the purposes they serve in the operating system, timer cells are assigned specific *roles* in SLOTH ON TIME—including activation cells, control cells, deadline cells, sync cells, aperiodic cells, and budget cells. Table 4.6 shows an overview of the assigned roles together with a short description of their purposes within the operating system; Figure 4.9 introduces the SLOTH ON TIME design for the example time-triggered application. Both the design and the individual cell roles are explained in detail in the following sections. The description assumes that the timer cells feature a counter register that is *incremented* upon each clock tick, generating an interrupt if the new value matches the one configured in the corresponding compare register. Furthermore, the cell can be controlled by enabling or disabling its interrupt request enable bit, which does not affect its counter handling, however.

[10]Note that a tick-based timer is used to implement the generalized time-triggered model and is not to be confused with the frame-based model, which also uses interrupt "ticks" at the start of each frame.

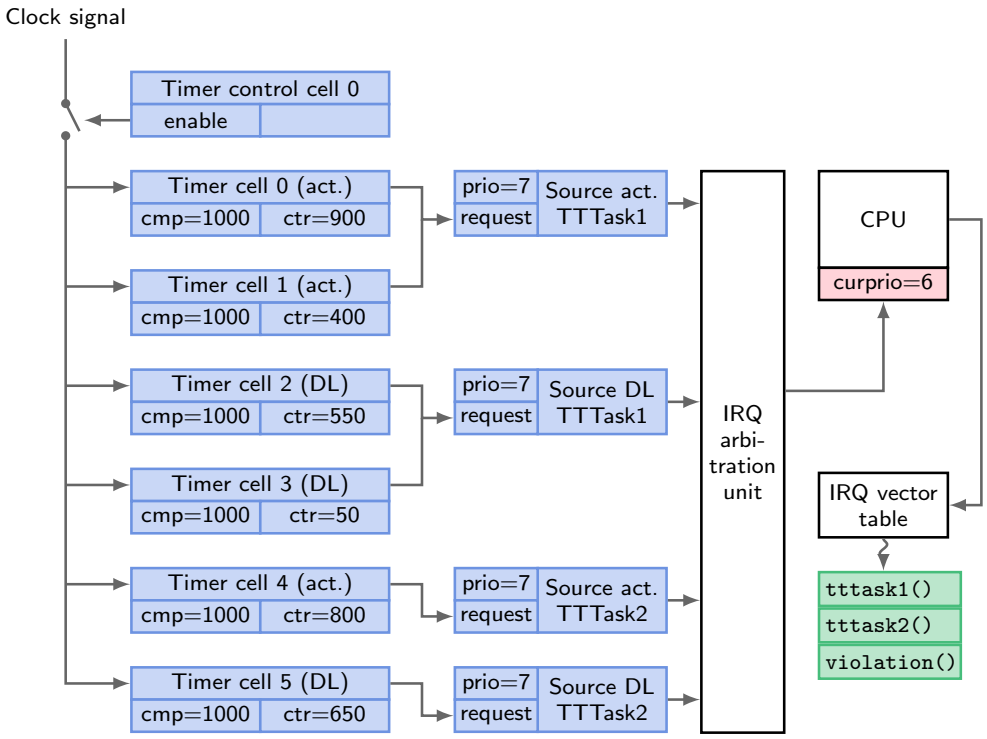


Figure 4.9: Design of the SLOTH ON TIME operating system. The design shows a SLOTH ON TIME system tailored to the example time-triggered application introduced in Table 4.5. Timer cells and interrupt sources are used for task activations (act.) and deadlines to be monitored (DL).

4.4.3 Time-Triggered Task Activations

SLOTH ON TIME adapts to the application by assigning each activation point in the dispatcher table a timer cell—making it an *activation cell* (see Table 4.6 and Timer Cells 0, 1, and 4 in Figure 4.9). SLOTH's hardware-centric approach allows for this design on platforms with an appropriate number of timer cells; if fewer cells are available, the design adapts to the platform by resorting to partial cell multiplexing, which is detailed in Section 4.4.8. In a next step, SLOTH ON TIME provides tailored initialization functionality for the involved timer cells (see Figure 4.10). The compare values for all cells are set to the length of the dispatcher round so that the cell generates an interrupt and resets the counter to zero after one full round has been completed. The initial counter value of a cell is set to the round length minus the expiry point offset in the dispatcher round. This way, once the cell is enabled, it generates its first interrupt after its offset is reached, and then each time a full dispatcher round has elapsed. In Figure 4.9, for instance, Timer Cell 4 is pre-configured to a compare value of 1,000 (the round length) and a counter value of 800 so that the first activation interrupt is trig-

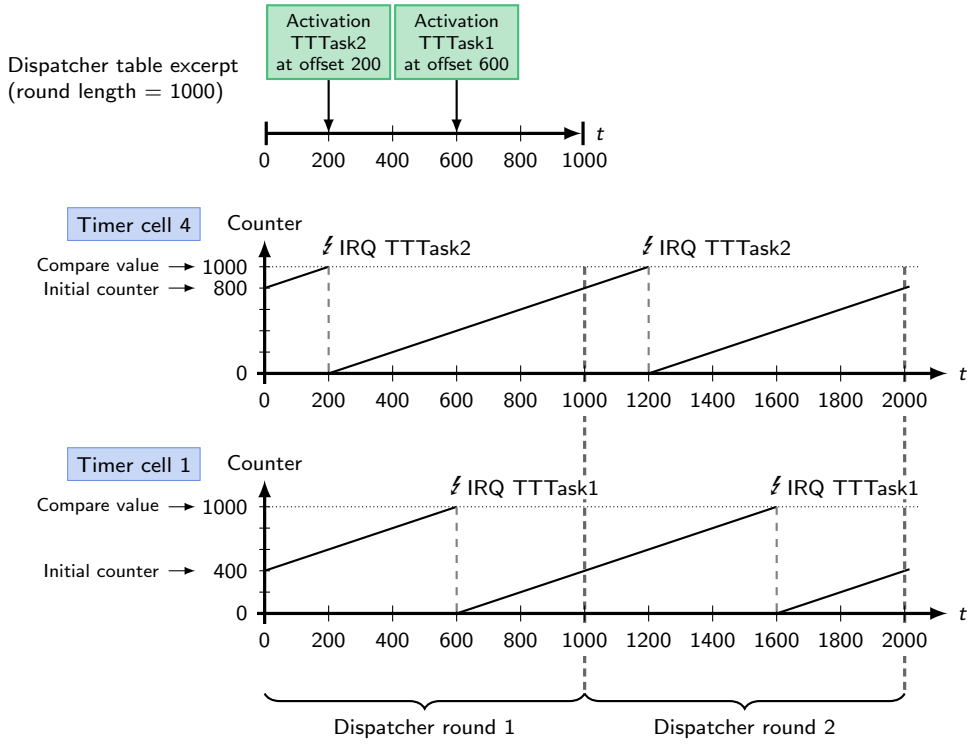


Figure 4.10: Instrumentation of timer cell compare values and initial counter values in SLOTH ON TIME. The figure shows an excerpt of the example dispatcher round in Figure 4.8 and the design in Figure 4.9, and the way the timer cell configuration works to generate interrupts at the appropriate points in time. For instance, by using a compare value of 1000 and an initial counter value of 800 in Timer Cell 4, an interrupt to dispatch `TTTask2` is generated at offset 200 of every dispatcher round.

gered after the offset of 200 for `TTTask2` (see configuration in Table 4.5 and run time behavior in Figure 4.10).

Additionally, SLOTH ON TIME provides tailored functionality for starting the dispatcher, which can take one of two forms depending on the underlying hardware platform. The hardware-centric design approach suggests that if the platform features hierarchically connected cascading timer cells, then a higher-order cell can be used as a so-called *control cell* to switch all connected lower-order timer cells on or off simultaneously (see Table 4.6 and Timer Control Cell 0 in Figure 4.9). If such a control cell is available, enabling it will enable all connected activation cells of a dispatcher round. This mechanism enables completely accurate and atomic starting and stopping of dispatcher rounds by setting a single bit in the respective control cell. If the platform does not support hierarchical cells, then the default implementation for starting the dispatcher enables all involved activation cells consecutively; the offsets need to be adapted correspondingly to

accommodate the additional latency, which is specific to the hardware platform and can be supplied by the operating system.

The SLOTH principle to extract as much information from the application configuration as possible in order to tailor static functionality to be executed once at *initialization time* leads to little or no logic for the dispatcher to cause overhead at *run time* during the productive phase of the system. The only exception is a mode change to have the system execute a different dispatcher round; in that case, the initialization overhead is incurred again for the mode change itself, but only once. At run time, no expiry points and dispatcher tables need to be managed by the operating system, since all required information is encapsulated in the timer cells that are pre-configured during the system initialization. Once the dispatcher is started by enabling the control cell or the individual timer cells, the interrupts corresponding to task dispatches will automatically be triggered at the specified expiry points by the timer system. The hardware interrupt system will then interrupt the current execution, which will either be the system's idle loop or a task about to be preempted, and it will dispatch the associated interrupt handler, which in SLOTH ON TIME basically corresponds to the task function as specified by the user, surrounded by a small wrapper (see vector table in Figure 4.9).

The only functions that are not performed automatically by the hardware and therefore need to be performed by the SLOTH ON TIME task wrapper are saving the full preempted task context when a new time-triggered task is dispatched and lowering the CPU priority from the interrupt *trigger priority* to the *execution priority*. This lowering is needed to achieve the stack-based preemption behavior of tasks in the system, such as proposed by the OSEKtime OS specification [OSE01], for instance. By configuring all interrupts to be triggered at a high trigger priority and lowering interrupt handler execution to a lower execution priority, every task can be preempted by any task that is triggered at a later point in time, yielding the desired stack-based behavior. In Figure 4.9, the IRQ sources are pre-configured to a high trigger priority of 7, whereas the task wrappers lower the priority to the execution priority of 6 (see current CPU priority). Thus, a task activation with a later expiry point implicitly has a higher priority than any preceding task activation. Additionally, SLOTH ON TIME registers the corresponding task functions for the task activation IRQ sources. In Figure 4.9, for instance, the interrupt handlers for the activation sources for `TTTask1` and `TTTask2` are set to `tttask1()` and `tttask2()`, respectively.

4.4.4 Deadline Monitoring

Deadlines to be monitored for violation are implemented in SLOTH ON TIME much in the same way that task activation expiry points are. Every deadline specified in the application configuration is assigned to a *deadline cell* (see Table 4.6 and Timer Cells 2, 3, and 5 in Figure 4.9). That timer cell is pre-configured to be triggered after the deadline offset, and then after one dispatcher round has elapsed;

similar to activation cells, this is reached by setting the compare and counter registers accordingly. The interrupt handler that is registered for such deadline cells is an exception handler for deadline violations that calls an application-provided handler to take action (see `violation()` handler in Figure 4.9).

In contrast to traditional implementations, SLOTH ON TIME disables the interrupt requests for a deadline cell once the corresponding task has run to completion, and re-enables them once the task has started to run. This way, deadlines that are *not* violated do not lead to unnecessary IRQs, which would disturb the execution of other real-time tasks in the system. In the system, this behavior is implemented by enhancing the task wrappers of the time-triggered tasks; here, the request enable bits of the associated deadline cells are enabled before the task is executed, and they are disabled after the task execution.

4.4.5 Synchronization with a Global Time Base

Real-time systems are rarely deployed stand-alone but often act together as a distributed system. Therefore, in time-triggered systems, synchronization of the system nodes with a global time base in the network needs to be maintained. This needs to be supported by the time-triggered operating system by adjusting the execution of dispatcher rounds depending on the detected clock drift.

If support for synchronization is enabled, SLOTH ON TIME allocates a dedicated *sync cell* (see also Table 4.6) and configures its offset to the point after the last deadline in a dispatcher round (in the example application configured in Table 4.5, this point would be 950). The amount of time between this point and the end of the dispatcher round can then be used to counteract a negative drift. Positive drifts are, of course, not restricted in this way, and can always be accommodated by the mechanism.

The interrupt handler attached to the sync cell then checks at run time—once per dispatcher round—whether a drift has occurred. If so, it simply modifies the counter values of all activation cells, deadline cells, and sync cells that belong to the dispatcher table, corrected by the drift value. Since the cell counters are modified *sequentially*, the last counter of a table is changed later than its first counter. However, since the read–modify–write cycle of the counter registers always takes the same amount of time and the modification value is the same for all counters, in effect it does not matter when exactly the counter registers are modified. In case the synchronization handler is not able to re-program all affected cell counters by the next task activation point in the schedule, it resumes execution at the next cell to be re-programmed once it becomes active again in the next round.

Figure 4.11 shows an example for how a positive drift is adjusted by the SLOTH ON TIME mechanism. After a dispatcher round with two activations, the synchronization timer IRQ detects a positive drift, which makes the synchronization handler adjust the counters of both activation cells and the sync cell itself. In effect, the next dispatcher round will be delayed by exactly the drift value.

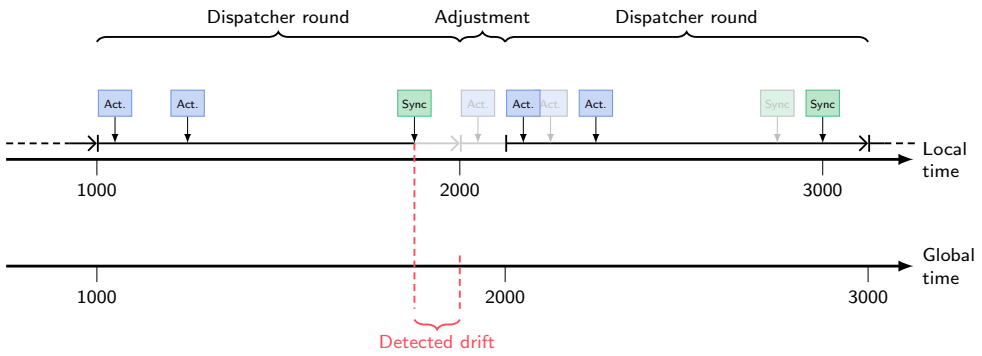


Figure 4.11: Example for time synchronization in SLOTH ON TIME. If a time drift is detected by the sync cell, it adjusts all dispatcher round cells by the detected offset to adjust the local time to the global time.

4.4.6 Combination of Time-Triggered with Event-Triggered Systems

Besides periodic tasks, which are perfectly accommodated by the time-triggered model, some situations require the handling of aperiodic events. One approach is to model this in so-called *periodic servers* [Liu00], which are described in this section. In the OSEK and AUTOSAR automotive standards used as examples for the considerations of this thesis, there are three additional approaches to combine event-triggered elements with a time-triggered system as implemented by SLOTH ON TIME. First, OSEKtime features controlled ISRs for aperiodic events; the interrupt sources are enabled only in pre-configured intervals in a dispatcher round. Additionally, the OSEK specifications describe the possibility of executing a mixed-mode system running an event-triggered OSEK OS system in the background time of the time-triggered OSEKtime OS system running on top, whereas the AUTOSAR OS standard specifies a system with event-triggered tasks that can optionally be activated at certain points in time using a schedule table abstraction.

Polling Servers

One approach to handle non-periodic events in time-triggered systems is to define a polling server task, which is allocated execution time in the time-triggered schedule as any other periodic task [Liu00]. The polling server is also allocated a specific execution budget, which is replenished regularly at the beginning of its execution. As long as the server has a positive execution budget during its execution at run time, it can dispatch sporadic and aperiodic tasks from a queue, preempting them when the budget is exhausted in order not to affect periodic task execution. This concept can be deployed together with the SLOTH ON TIME operating system since it is based on scheduling a periodic application task as a "proxy task".

In different varieties, real-time literature presents concepts of bandwidth-preserving servers, which include deferrable servers [SLS95] and sporadic servers [SSL89]. These servers differ in when the execution budget expires upon not claiming it entirely and when and how exactly the budget is replenished; their goal is to improve the response times of the aperiodic tasks compared to the polling server. Support by the SLOTH ON TIME operating system for those concepts can be provided by mapping dedicated timer cells if available on the hardware platform. This way, the server does not necessarily have to use a software-multiplexed hardware timer and counter in order to maintain its execution budget (see also Section 4.4.7).

Interrupt Service Routines for Aperiodic Events

OSEKtime also specifies application ISRs to accommodate aperiodic events in the time-triggered schedule. These ISRs have an execution priority that is higher than the time-triggered tasks in the system; SLOTH ON TIME therefore assigns interrupt priorities higher than its trigger priority to the corresponding interrupt sources.

In order to bound the effects of ISR occurrences on the deterministic time-triggered schedule, the operating system needs to allow an aperiodic interrupt only in a pre-allocated interval of a dispatcher round and restrict it to one execution instance per interval. In SLOTH ON TIME, one *aperiodic cell* is therefore allocated per aperiodic event (see also Table 4.6) and pre-configured to trigger at the configured ISR enable time in the dispatcher round. When the aperiodic cell triggers, a short timer handler enables the aperiodic interrupt source. The aperiodic ISR itself is contained in a wrapper, which disables the corresponding interrupt source after the execution of the ISR to guarantee the single execution within one interval. The responsibility for accommodating those periods of aperiodic interrupts in the time-triggered schedule without affecting deadline-aware execution of the periodic time-triggered tasks is with the real-time application developer, who can use external tool support for constructing appropriate schedules.

Mixed-Mode System

Since the SLOTH operating system implements an event-triggered interface, whereas SLOTH ON TIME implements a time-triggered interface, both systems can be combined simply by separating their priority spaces by means of configuration. By assigning all event-triggered tasks priorities that are lower than the time-triggered execution and trigger priorities, the event-triggered system is only executed when there are no time-triggered tasks running—that is, in the background time of the time-triggered system. Additionally, the event-triggered SLOTH operating system synchronization priority, which is used to synchronize access to operating system state against asynchronous task activations (see Section 5.5.5), is set to the highest priority of all event-triggered tasks but lower than the time-

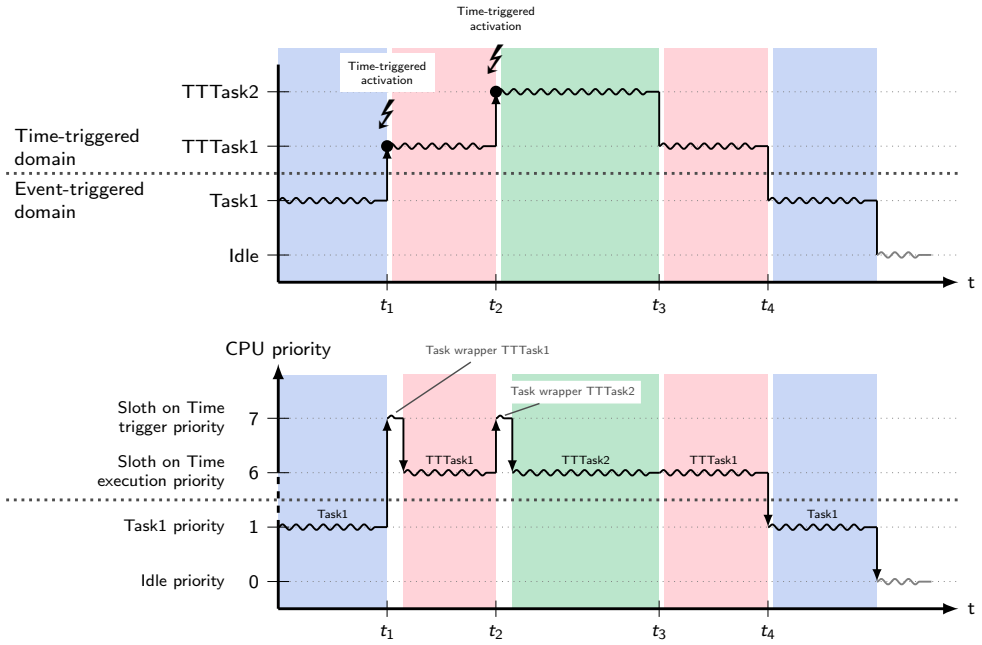


Figure 4.12: Example control flow trace in a mixed-mode event-triggered/time-triggered and SLOTH/SLOTH ON TIME system. The top part of the figure shows the abstract specification as stated by the OSEK OS and OSEKtime OS standards used as an example; the lower part shows the implementation in SLOTH and SLOTH ON TIME using separation of event-triggered and time-triggered priority spaces.

triggered priorities. This way, the event-triggered system can be preempted by a time-triggered interrupt at any time.

Thus, the integration of both kinds of systems can easily be achieved without jeopardizing the timely execution of the time-triggered tasks. The SLOTH and SLOTH ON TIME designs running tasks with interrupt priorities are perfectly suited for dealing with events in time-triggered systems; the events are processed automatically in the background time of the time-triggered system.

Figure 4.12 shows an example control flow in a mixed-mode system with an event-triggered SLOTH application (such as the one presented in Table 4.1) running in the background time of a time-triggered SLOTH ON TIME application (such as the one presented in Table 4.5). The event-triggered application can be interrupted by time-triggered IRQs at any time, since it uses priorities between 1 and 4, whereas the SLOTH ON TIME execution and trigger priorities are set to 6 and 7, respectively.

Event-Triggered System with Time-Triggered Elements

In contrast to the mixed-mode approach, AUTOSAR OS defines an event-triggered operating system with static task priorities; its schedule table abstrac-

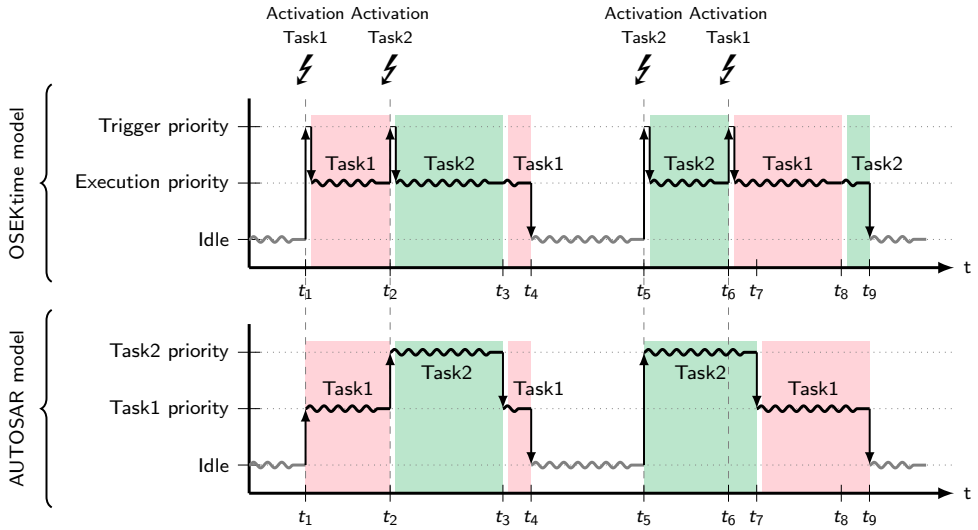


Figure 4.13: Time-based execution models in OSEKtime OS and AUTOSAR OS. The figure shows the way that SLOTH implements the OSEKtime-like generalized time-triggered execution model (top) and the AUTOSAR OS execution model (bottom). Note that the second activation of `Task1` at t_6 does not lead to a dispatch in AUTOSAR OS due to its lower priority compared to `Task2`.

tion only provides means to *activate* the event-triggered tasks at certain points in time, inserting them into the operating system list of tasks that are ready for execution. Such a time-based task activation does not necessarily lead to a *task dispatch* as in purely time-triggered systems; the dispatch is deferred when a higher-priority task is currently running. In contrast to time-triggered tasks, AUTOSAR OS tasks have application-configured and potentially distinct priorities; at run time, they can additionally raise their execution priority by acquiring resources for synchronization or even block while waiting for an event. The difference at run time compared to a generalized time-triggered system is sketched in Figure 4.13. If `Task1` is activated before `Task2` (left part of the figure), both kinds of systems effectively show the same behavior. If, however, `Task1` is activated *after* `Task2` in the AUTOSAR system with `Task2` still running (right part of the figure), this activation does not lead a dispatch due to its lower priority (lower part of the figure); in OSEKtime-like generalized time-triggered systems, the stack-based behavior requires the currently running task to be preempted in any case (upper part of the figure). In addition to supporting the time-triggered model as described so far, the SLOTH ON TIME prototype also supports the AUTOSAR model of time-based activations using the schedule table abstraction.

The schedule table abstraction only adds a time-based element to the event-triggered SLOTH execution model, but it is implemented in SLOTH ON TIME in a way that is very similar to the time-triggered dispatcher table presented in Sec-

tion 4.4.3. Instead of configuring the priorities of the IRQ sources attached to the timer system to the system trigger priority (see Figure 4.9), however, they are set to the priority of the task they activate. This way, the time-dependent activation of tasks is seamlessly integrated into the execution of the rest of the SLOTH system, since after the IRQ pending bit has been set, it does not matter whether this was due to a timer expiry or due to a synchronous task activation system call. This is similar to the way that SLOTH alarms are designed (see Section 4.2.4).

To fully implement schedule tables according to the example requirements of the AUTOSAR OS specification, the SLOTH ON TIME timer facility is enhanced in three ways. First, AUTOSAR allows multiple schedule tables to be executed simultaneously and starting and stopping them at any time. Thus, SLOTH ON TIME introduces the corresponding system calls `StartScheduleTable()` and `StopScheduleTable()`, which enable and disable the control cell for the corresponding schedule table, respectively. Second, AUTOSAR defines *non-repeating* schedule tables, which encapsulate expiry points for a single dispatcher round, executed only once when that schedule table is started. SLOTH ON TIME implements this kind of schedule table by pre-configuring the corresponding timer cells to one-shot mode; this way, they do not need to be manually de-activated at the end of the schedule table. Third, schedule tables can be started with a time offset specified dynamically at run time as a system call parameter; this time offset delays the execution of the whole schedule table by the specified amount of time in the operating system. In that case, SLOTH ON TIME re-configures the statically configured timer cells for that schedule table to include the run time parameter in its offset calculation before starting it by enabling its control cell.

Since time-based task activations occur directly at the corresponding task priority in SLOTH ON TIME, they are correctly prioritized; high-priority tasks are not interrupted by low-priority task activations as in traditional systems. This way, *by design*, SLOTH ON TIME prevents indirect rate-monotonic priority inversion in the system.

4.4.7 Execution Time Protection

For isolation of faults that are caused by software bugs or external influences that make tasks execute longer than their statically defined worst-case execution times, time-triggered systems use frame overrun detection or deadline monitoring (see Section 4.4.1 and Section 4.4.4). The event-triggered AUTOSAR standard, which is again used as a requirements example here, instead prescribes timing protection facilities using execution time budgeting to restrict the effects of such misbehaving tasks on the whole system. Each task is assigned a maximum execution budget per activation, which is decremented while that task is running, yielding an exception when the budget is exhausted. In its design, SLOTH ON TIME employs the same mechanisms used for expiry points and deadlines to implement those task budgets. It assigns one *budget cell* to each task to be monitored (see Table 4.6), initializes its counter with the execution time budget provided by the

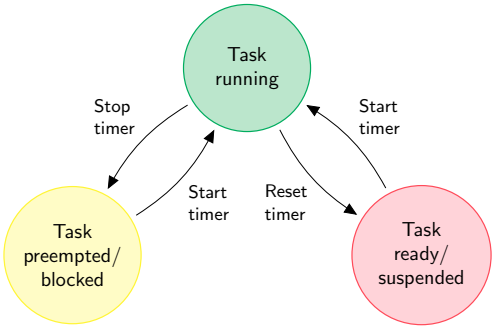


Figure 4.14: SLOTH ON TIME task states and budgeting transitions. Upon each task transition, SLOTH ON TIME uses the corresponding timer cells for execution time protection of the involved tasks.

application configuration, and configures it to run down once started. The associated IRQ source is configured to execute a user-defined protection hook as an exception handler in case the budget cell timer should expire.

Furthermore, the dispatch points in the system are instrumented to pause, resume, and reset the budget timers appropriately (see task states and transitions in Figure 4.14). First, this entails enhancing the prologue part of the task wrappers, which pauses the budget timer of the preempted task and starts the budget timer of the task that is about to run. Second, the epilogue part of the task wrapper, which is executed after task termination, is added instructions to reset the budget to the initial value configured for this task, and to resume the budget timer of the previously preempted task whose context is about to be restored. This design allows for light-weight monitoring of task execution budgets in event-triggered systems at run time without the need to maintain and calculate using software counters; this information is implicitly encapsulated and automatically updated by the timer hardware in the counter registers.

Figure 4.15 shows an example control flow in a system with task budgets. Both `Task1` and `Task2` have execution budgets of 400; when `Task1` is started at 200 (point ①) in the figure), its budget starts running down as long as it is running. When `Task1` is preempted by `Task2` at 400 (point ②), the `Task2` prologue wrapper stops the budget timer of the preempted `Task1` and starts its own budget timer to run down. When terminating at 700 (point ③), the `Task2` epilogue wrapper resets its own budget timer to the initial value of 400 for the next execution instance, and it starts the budget timer for the task about to run, `Task1`, restoring its remaining budget of 200. At 900 (point ④), that remaining budget is exhausted, so the budget cell automatically triggers an interrupt, which is handled by a user-provided protection violation handler.

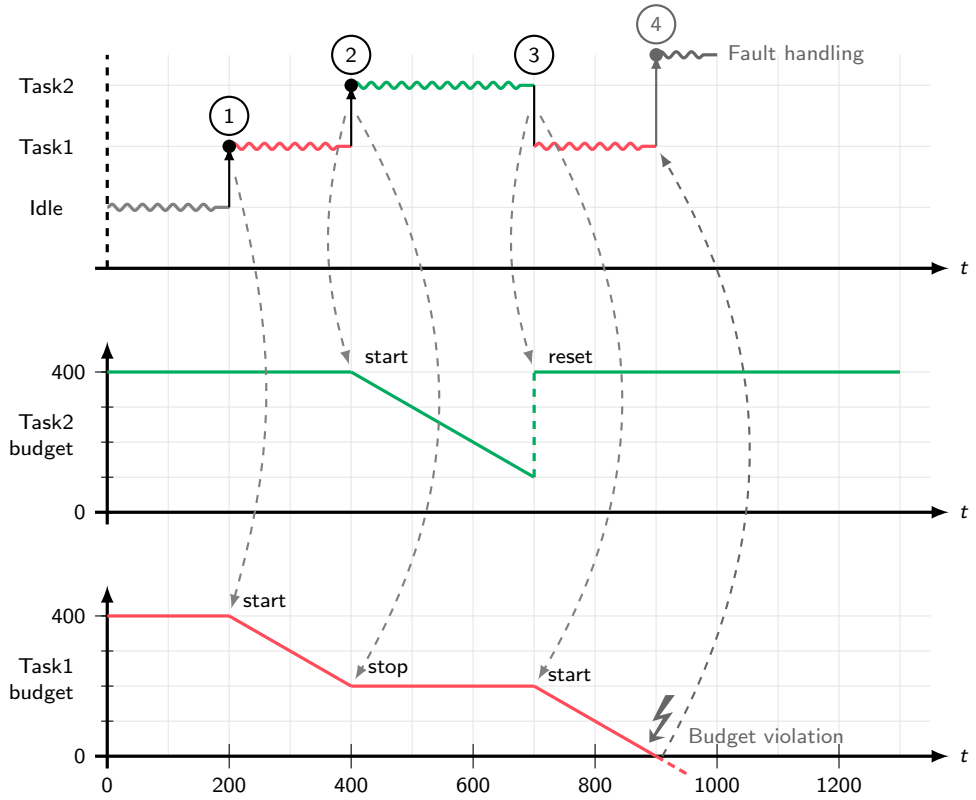


Figure 4.15: Example control flow trace in SLOTH ON TIME with execution budgeting. The figure shows an application with two tasks and the corresponding trace of the task budgets. `Task1`'s budget is exhausted at 900, which leads to a budget violation fault.

4.4.8 Timer Cell Multiplexing

If the hardware platform does not have enough timer cells to allocate one cell per time-triggered event, SLOTH ON TIME also allows to partially fall back to multiplexing due to its hardware-centric design. In that case, it allocates only one timer cell for each role (activation and deadline) *per task* and partly re-configures it at run time.

For multiplexed deadline monitoring, if the current deadline has not been violated, the epilogue part of the task wrapper re-configures the expiration point of the deadline cell to the *next* deadline instead of disabling it. This way, the non-violated deadline does not lead to an exception IRQ, but the next deadline is active again. The deltas between the deadline points are retrieved from a small offset array, which can be held in ROM.

For multiplexed activations of the same task, another offset array contains the deltas between the activation points of that task. The prologue part of the task wrapper is then enhanced to re-configure the compare value of the activation cell

to the next activation point every time that task is dispatched. This way, the *next* activation of the task is scheduled to be triggered at the right point in time.

4.4.9 SLOTH ON TIME Summary

SLOTH ON TIME implements a time-triggered real-time operating system by offering frame-based execution or by instrumenting multiple timer cells of the hardware timer system if the hardware platform supports that model. By pre-configuring timer cells to fire at task activation points according to the application-provided dispatcher table and linking them to task interrupt handlers, the operating system has little to do once the dispatcher is started. Deadline cells are used to dispatch an exception handler when a monitored deadline is violated; deadline cells are de-activated when a task has run to completion in time. By using a timer cell as a sync cell, clock drifts compared to a global time source can be accommodated by adjusted the counter values of the involved cells.

Timing-related concepts and abstractions in *event-triggered* operating systems can similarly be implemented using the SLOTH ON TIME approach, as can concepts to handle aperiodic events. Time-based task activations in event-triggered systems respect task priorities, and task execution budgets are automatically maintained by using timer cells as budget cells. Related work also optimizes the design of timer abstractions in the operating system; in contrast to SLOTH ON TIME, however, the corresponding approaches mainly focus on system-internal timer scheduling and management algorithms without a hardware-centric design (see discussion in Section 7.3.4).

Table 4.7 shows a summary of the way that the SLOTH ON TIME design implements time-triggered operating system features.

4.5 Design Summary

The design of the SLOTH embedded operating system focuses on utilizing available hardware subsystems for operating system purposes if the hardware platform permits (see Figure 4.16). It relies on the interrupt subsystem to do its control flow scheduling and dispatching, and it relies on the timer subsystem for scheduling *time-triggered* control flows. Since those subsystems operate concurrently to the main CPU, which executes the real-time application, SLOTH can thereby hide operating system latencies and increase the overall operating system performance as perceived by the application. The SLOTH design goal is based on moving operating system information from run time during the productive phase of the system to its initialization time, exemplified by pre-configuring both the interrupt subsystem with the application scheduling data (i.e., control flow priorities) and the timer subsystem with time-based scheduling data—when *initializing* the system once before starting the main execution. At run time, SLOTH systems can therefore minimize operating system software activity, focusing CPU

Operating system feature	SLOTH ON TIME design
Time-triggered task dispatch	Timer cell with pre-configured offset and cycle time; connected IRQ source with trigger priority and task function as interrupt handler.
Deadline monitoring	Timer cell with pre-configured offset and cycle time; activated and de-activated in corresponding task wrapper.
Time synchronization	Synchronization handler adjusts involved timer counters by the detected clock drift.
Mixed-mode system	Time-triggered SLOTH ON TIME system runs event-triggered SLOTH system in its background time by employing higher priorities than all SLOTH priorities.
Aperiodic ISRs	Timer enables aperiodic interrupt source in pre-configured interval; source is disabled in ISR epilogue.
Time-based event-triggered dispatch	Timer cell with pre-configured offset and cycle time; connected IRQ source with task priority as in SLOTH.
Execution time protection	Timer cell maintaining task budget; started, stopped, and reset when the task state changes.

Table 4.7: Summary of the way that the SLOTH ON TIME design implements embedded operating system features. SLOTH ON TIME implements time-triggered operating system features and time-based features for event-triggered operating systems. See also the SLOTH and SLEEPY SLOTH designs in Table 4.2 and Table 4.4, respectively.

time on the real-time application itself. Additionally, by moving all types of control flows to the interrupt processing level using the unified interrupt priority space, all SLOTH systems prevent rate-monotonic priority inversion *by design*.

The SLOTH systems feature a universal control flow abstraction, which is based on the interrupt service routine abstraction offered by the hardware and which is adapted with minimized operating system software to be tailored to the application-defined semantics. Using this configurable control flow abstraction, SLOTH can be used to design event-triggered systems with stack-based run-to-completion execution (SLOTH implementation) and event-triggered operating systems with blocking task semantics (SLEEPY SLOTH implementation). Additionally, the SLOTH ON TIME design is suitable to implement time-triggered operating systems in different variants, including frame-based and generalized time-triggered execution. All of the SLOTH task types and systems can be seamlessly combined in one aggregated system that exactly adheres to the semantics as defined by the real-time application. Such a mixed-mode system can flexibly be run as an event-triggered subsystem in the background time of the time-triggered one, for instance.

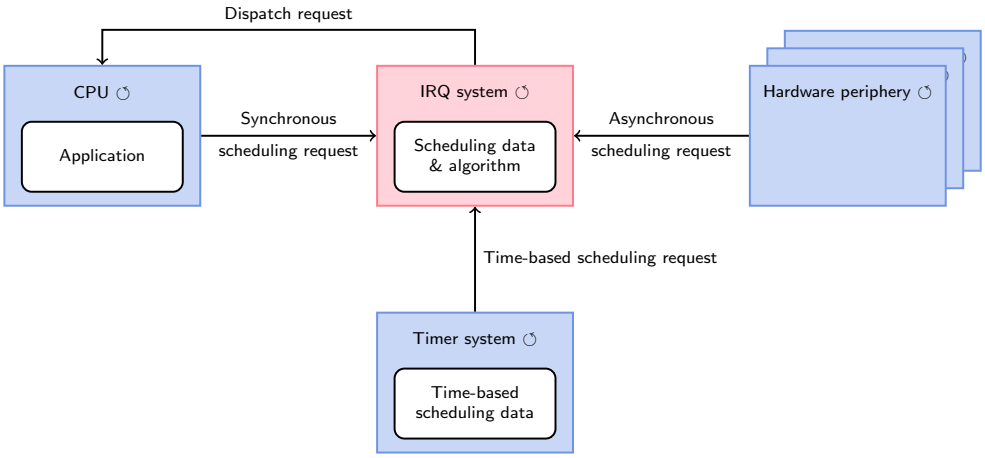


Figure 4.16: Overview of the way that SLOTH utilizes hardware execution units if available. These units operate concurrently on a typical hardware platform (see ↻ symbol) and contribute to the latency hiding property of the SLOTH operating systems this way.

Implementation 5

> Detective Lt. William Somerset (Morgan Freeman): There are seven deadly sins, Captain. Gluttony..., greed..., SLOTH, wrath, pride, lust, and envy. Seven.
>
> *From the movie* Seven*, 1995*

The SLOTH system is implemented in a way that reflects and supports the design decisions presented in the previous chapter. This chapter first details the SLOTH system framework in Section 5.1, together with its build process and file structure in Section 5.2 and in Section 5.3. After that, the hardware platform requirements for an ideal implementation of the hardware-centric SLOTH design are given in Section 5.4 before presenting the platform-specific implementation on the Infineon TriCore TC1796 reference platform in Section 5.5 and comparing it to another implementation on the Atmel SAM3U platform in Section 5.6.

5.1 Static Analysis and System Generation Framework

SLOTH, as most embedded systems, is implemented in the C programming language. This is due to the fact that only C compilers are available for virtually any embedded hardware platform, which are the main target of the SLOTH approach. Additionally, since the SLOTH approach is even closer to the hardware than other embedded-system designs, it depends to a higher degree on the ability of the corresponding C compiler to produce optimized machine code for the given hardware platform. Where possible, the SLOTH system code aids the compiler in that process by using programming language constructs that facilitate the compiler analysis. In particular, SLOTH expects from the compiler to perform constant propagation and code inlining in order to provide a lean machine code implementation for its system calls, which are implemented in several implementation layers in hardware-independent, hardware-specific, application-independent, and application-specific parts (see also Section 5.1.1). Since code inlining trades increased code size for decreased data size, this feature can be configured by the SLOTH application developer; SLOTH only provides a high *potential* and benefit for inlining if desired by the developer (see Section 6.2).

The SLOTH operating system targets a statically configured embedded application—and as such, the application also provides a configuration file in addition to its actual implementation. This configuration entails both the *system* configuration—that is, which features the operating system needs to provide for the application to run correctly—and the *application* configuration—that is, which kinds of operating system objects the application allocates and with which properties. Examples for the former class of configuration options are the need to support resource abstractions or the specification of fully preemptive scheduling, whereas examples for the latter class include the number of application tasks, their static priorities, and whether they are of basic or extended type.

In order to be able to tailor the SLOTH operating system to the application and the hardware, the framework uses a generative approach to provide for the system adaptation. Generation allows for arbitrarily fine-grained tailoring through the ability of adapting source code text, and it allows for generating code for an arbitrary number of instances of application objects. Both of those characteristics are needed by SLOTH, since SLOTH needs to adapt its source code to particular register names for a given hardware platform, for instance (hardware dimension of tailoring), and it needs to produce code for an arbitrary number of configured application objects, such as tasks and ISRs, for instance (application dimension of tailoring). A generative approach therefore provides the necessary level of flexibility needed to implement the two-dimensional tailoring of the SLOTH approach (see also Section 3.1).

On the meta level, any Turing-complete language is suitable to express the intent of the C code generation. SLOTH uses the Perl script language due to its versatility and high expressivity; additionally, generation loops for operating system objects can be expressed in a very concise manner using Perl `foreach` language constructs. Together with its ability to define flexibly nested data structures for configuration models, Perl therefore provides a very good basis for SLOTH to generate its configuration-dependent C code from. The generated code parts also include the flexible parts of the SLOTH flexible HAL (see also Section 3.3.2).

5.1.1 SLOTH Perl Configuration and Templates

In SLOTH, both kinds of configurations—the application configuration and the system configuration—are specified in a Perl-based domain-specific language in a `config.pl` file accompanying the C implementation files of the application. The configurations are specified as two Perl hash tables `$appconfig` and `$sysconfig`, which themselves include different hash tables that include the configuration parameters; using a hash data structure, the value corresponding to a (string) key can be retrieved in a single operation. The configuration file for the example application sketched in Table 4.1 is shown in Figure 5.1.

Since, besides being tailored to the hardware platform below, SLOTH is also tailored to the application above, parts of the operating system are generated depending on the system and application configuration, which includes the SLOTH

```
$appconfig = {
  'tasks' => {
    'Task1' => {
      'priority' => '1',
      'autostart' => 'true',
    },
    'Task3' => {
      'priority' => '3',
    },
    'Task4' => {
      'priority' => '4',
    },
  },
  'isrs' => {
    'ISR2' => {
      'priority' => '2',
      'source' => 'ASC0_RSRC',
    },
  },
  'resources' => {
    'Res1' => {
      'usedByTasks' => ['Task1', 'Task3'],
    },
    'Res2' => {
      'usedByTasks' => ['Task2', 'Task3'],
    },
  },
  'alarms' => {
    'Alarm1' => {
      'activates' => 'Task4',
    },
  },
};
```

```
$sysconfig = {
  'resourceSupport' => 'yes',
  'alarmSupport' => 'yes',
  'interruptSupport' => 'yes',
  'eventSupport' => 'no',
  'multipleActivationsSupport' => 'no',
  'hasStartupHook' => 'no',
  'preemption' => 'full',
  'tc1796_arbRounds' => '2',
  'tc1796_busCyclesPerArbRound' => '1',
  'tc1796_systemFrequency' => '5000000',
};
```

Figure 5.1: Example SLOTH application configuration and system configuration. The application configuration for the application introduced in Table 4.1 is shown in the left part, a corresponding system configuration is shown in the right part of the figure. Both types of configurations are implemented in SLOTH as hash tables in a Perl file `config.pl` provided with the application implementation.

flexible HAL (see also Section 3.3.2). For instance, the code initializing the interrupt sources processes the task priorities and auto-start properties as specified in the application configuration. Since the configuration is implemented in hash tables in Perl syntax, it can directly be processed by template files implemented in Perl to produce those generated operating system parts. The template files describe a C implementation file depending on the configuration; they can contain arbitrary Perl code that outputs C code (see flexible HAL instance in Figure 3.3).

Figure 5.2 shows an overview of the two dimensions that the SLOTH operating system is tailored to; one dimension distinguishes between hardware-independent and hardware-specific parts, whereas the second dimension distinguishes between application-independent and application-specific parts. Only the application/system configuration and the application itself need to be supplied by

the application programmer. The application-specific files that are actually processed by the compiler are generated from the template files. As an example, Figure 5.3 shows the hardware-specific triggering of interrupt sources for task activation on the Infineon TriCore platform—both its template and the generated part fed to the compiler. Since generated functions that include such if–else cascades depending on an involved operating system object are called by SLOTH only with static object parameters, SLOTH aids the compiler in statically optimizing such functions to a single instruction. In this case, a store machine instruction triggers the IRQ for the corresponding task in the memory-mapped IRQ source register. Other occurrences of such optimizable if–else cascades include the functions to clear a pending interrupt and to enable or disable an IRQ source to block or unblock a task in SLEEPY SLOTH.

Application-specific and therefore generated parts of the SLOTH operating system include, amongst others (see also Figure 5.2),

- the instantiation of operating system objects such as named task identifiers (IDs), resource IDs, etc., and associated data such as task stacks for extended tasks;
- the system configuration of the operating system as C pre-processor variables (e.g., if `resourceSupport` is set to `yes` in the configuration, the pre-processor variable `CONFIG_RESOURCES` is defined);
- the IRQ vector table, including the SLOTH task functions, prologues, and exception handlers in the table to be jumped to after the corresponding IRQ has been triggered;
- initialization code for the involved IRQ sources, which sets the IRQ priorities to the task priorities (in event-triggered systems) or to the trigger priority (in time-triggered systems) and the pending bits depending on the auto-start parameters in the configuration;
- management code for the involved IRQ sources, which sets the pending bits depending on the task parameter, for instance (see also Figure 5.3);
- and initialization and management code for time-triggered schedule tables, pre-configuring and re-configuring the timer cells according to the configuration.

The SLOTH flexible HAL instance (see Figure 3.3) effectively consists of the architecture-specific templates in Figure 5.2, whereas the residual HAL program consists of the depicted application-specific and architecture-specific files. The hardware specialization of the flexible meta HAL to the flexible HAL instance is performed by having the build system select the appropriate architecture templates in the build process.

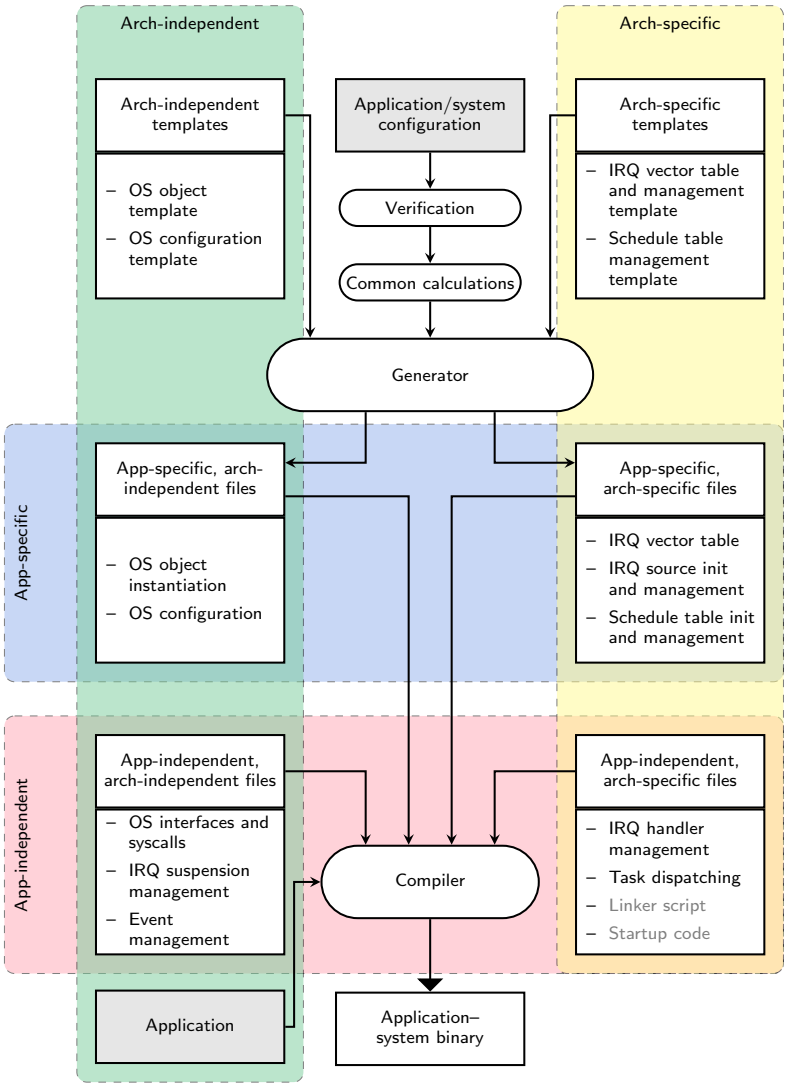


Figure 5.2: Overview of the two dimensions that the SLOTH operating system is tailored to. Application-specific parts are generated depending on the configuration and compiled together with application-independent parts of the operating system. Both parts are clearly subdivided into architecture-specific and architecture-independent parts to facilitate porting to another hardware platform.

Using this configuration and generation mechanism, the SLOTH application is tailored to the universal control flow abstraction used by the SLOTH system internally (see also Section 4.1). The programming model, on the other hand, stays the same for the application and is only adapted by SLOTH according to the system configuration also provided by the application. Thus, the application can rely on

```perl
print "ALWAYS_INLINE void tc1796Setr(TaskType task) {\n";
my $tasklines = "\t";
foreach my $taskname (sort(keys(%{$appconfig->{'tasks'}}))) {
  my $srn = $appconfig->{'tasks'}->{$taskname}->{'tc1796_srn'};
  $tasklines .= "if (task == $taskname) $srn.bits.SETR = 1;\n" . "\telse ";
}
$tasklines =~ s/\telse $//; # chop last else
print $tasklines . "}\n";
```

(a) Perl template code for triggering an interrupt source for a task.

```
ALWAYS_INLINE void tc1796Setr(TaskType task) {
  if (task == Task1) CPU_SRC0.bits.SETR = 1;
  else if (task == Task3) CPU_SRC1.bits.SETR = 1;
  else if (task == Task4) CPU_SRC2.bits.SETR = 1;
}
```

(b) Example generated C code for triggering an interrupt source for a task.

Figure 5.3: Example SLOTH Perl template file and generated C code. The example shows (a) the template for the main part of activating a task by triggering its interrupt source on the Infineon TriCore TC1796 and (b) the generated C code for the example application introduced in Table 4.1 and its Perl configuration in Figure 5.1. Since the generated function is only called with static task parameters, the compiler can statically optimize the if–else cascade to a single instruction.

the same stable interfaces and is shielded from SLOTH-internal implementation particularities (see also Section 4.1).

5.1.2 Configuration Verification and Analysis

Before applying the application and system configuration to the template files, the SLOTH generator first verifies it for obvious misconfigurations and analyzes it to produce information commonly used by several template files and to further optimize the resulting system (see also Figure 5.2).

The verification step is implemented in a Perl module `verify.pm`, which is evaluated by the generator after evaluating the configuration as provided by the application. The verification module includes sanity checks that access configuration parameters and that output warnings or errors, depending on the severity of the failed check. In generalized time-triggered SLOTH ON TIME systems, for instance, the verification includes checks that all defined deadlines belong to a preceding task activation in a dispatcher round; otherwise, an error is thrown, and the system generation fails.

The analysis step calculates common information needed by the templates, and it stores this information directly in the hash tables alongside the configured parameters (see also Figure 5.2). The template files can then transparently access all information, independent of whether it is configured by the applica-

tion or subsequently deduced by the generator. Calculated information includes, amongst others,

- the calculation of the ceiling priorities of the involved resources, depending on the priorities of the tasks configured to access them;
- the assignment of dedicated IRQ sources to resources in SLEEPY SLOTH systems (see Section 4.3.6) from a pool of available sources specified in the configuration;
- the calculation of the maximum priority of all tasks in the system, needed for the system synchronization and the scheduler lock resource `RES_SCHEDULER` (see Section 4.2.5), amongst others;
- the calculation of the trigger and execution priorities in mixed-mode SLOTH ON TIME systems (see Section 4.4.6), depending on the involved event-triggered priorities;
- hardware-specific timing information for synchronization with the interrupt arbitration system and its induced latencies (e.g., the number of `nop` instructions to include depending on the system frequency and the involved IRQ priorities on the Infineon TriCore TC1796; see Section 5.5.1).

In addition to deducing information from the configuration, the SLOTH generator also uses the provided information about the static nature of the application to further optimize the generated system. Such optimizations include:

- The generator analyzes the application priorities as specified by the configuration and maps the logical priorities as provided by the real-time application to physical IRQ priorities on the hardware platform. In this step, the physical priority space can be compacted if the logical priority space is sparse. Additionally, dedicated priority slots are allocated in the physical priority space to unambiguously identify resources in SLEEPY SLOTH systems (see Section 4.3.6).
- In SLEEPY SLOTH systems with both extended and basic tasks in the application, the generator analyzes the task priorities and properties to deduce which tasks and ISRs can actually be preempted by which other ISRs and tasks at run time. This way, the parts in the task prologues that check for the type of the preempted task to perform a full context switch with a stack switch or not can be omitted for some of the tasks. This is the case if a basic task can only preempt other basic tasks, for instance. Thus, SLEEPY SLOTH tries to evaluate those conditions statically at compile time where possible and dynamically at run time otherwise.

5.1.3 SLOTH ON TIME Timer Cell Mapping

In event-triggered SLOTH and SLEEPY SLOTH systems, the analysis and generation part mainly takes care of the mapping of tasks to interrupt sources and the corresponding hardware control routines. In time-triggered SLOTH ON TIME systems, an additional component that needs to be managed is the timer cell array that is used for time-triggered task dispatching. SLOTH ON TIME applications provide a configuration that differs from the one provided with regular SLOTH applications (compare Figure 5.4 to Figure 5.1). Its configuration comprises the specification of the timing parameters of the dispatcher table and a description of the timer hardware to be used by the SLOTH ON TIME operating system.

The SLOTH ON TIME generator takes this configuration as its input, and, in an intermediate step, calculates a mapping of the included expiry points to individual activation cells, which are subject to platform-specific mapping constraints due to the way that the individual timer cells are interconnected. This platform-specific information is contained in the generator and includes the platform mappings of timer cells to IRQ sources and the information which control cells are connected to which other timer cells if available. The timer cells for SLOTH ON TIME to use are taken from a pool of cells marked as available by the application; this information is also provided by the application configuration (see `tc1796_availableTimerCells` in Figure 5.4). If the number of available timer cells is not sufficient, the SLOTH ON TIME generator falls back to partial multiplexing of timer cells (see Section 4.4.8).

The right part of Figure 5.4 shows the mapping for the example application on the TriCore TC1796. Due to a platform peculiarity, two adjacent timer cells are needed per activation or deadline check. The same IRQ source can be used for an activation or deadline check of the same task (see IRQs 24 and 25). On the TC1796, the timer cell GTC08 can be used as a control cell for the involved timer cells of the configured application; this way, all cells can be started and stopped simultaneously (see also Section 4.4.3).

In the next step, the calculated mapping is used to generate initialization code for the involved timer cells, including the compare and initial counter values (see also Section 4.4.3). Additionally, the code to start the dispatcher round is generated using the corresponding control cell if available. Furthermore, code to initialize the IRQ system with the SLOTH ON TIME trigger priorities and to register the IRQ handlers with the task functions for activation cells and with the deadline exception handler for deadline cells is generated (see also Section 4.4.3 and Section 4.4.4). Eventually, the generator outputs the prologue and epilogue wrapper code per time-triggered task (see example in Figure 5.5). The prologue code saves the full context of the preempted task, enables the IRQ requests for the deadline cells for the corresponding task, and lowers the CPU priority from the SLOTH ON TIME trigger priority to the SLOTH ON TIME execution priority.[1] The epilogue code disables the IRQs for the corresponding deadline cells (since the

[1] On the TC1796 platform, the deadline at 450 needs two adjacent timer cells, 12 and 13, but

```
$appconfig = {
  'tasks' => {
    'TTTask1' => {
      'type' => 'osektt',
    },
    'TTTask2' => {
      'type' => 'osektt',
    },
  },
  'scheduleTables' => {
    'SchedTab1' => {
      'duration' => '1000',
      'repeating' => 'true',
      'autostart' => 'relative',
      'autostartValue' => '3000',
      'expiryPoints' => {
        '100' => {
          'type' => 'activation', 'task' => 'TTTask1',
        },
        '450' => {
          'type' => 'deadline', 'task' => 'TTTask1',
        },
        '200' => {
          'type' => 'activation', 'task' => 'TTTask2',
        },
        '350' => {
          'type' => 'deadline', 'task' => 'TTTask2',
        },
        '600' => {
          'type' => 'activation', 'task' => 'TTTask1',
        },
        '950' => {
          'type' => 'deadline', 'task' => 'TTTask1',
        },
      },
    },
  },
  'tc1796_availableTimerCells' => [8 .. 15, 40 .. 47],
};
```

```
Activation of TTTask1 at 100
    -> Timer Cell 08 and 09, IRQ 24
Activation of TTTask1 at 600
    -> Timer Cell 10 and 11, IRQ 24
Deadline of TTTask1 at 450
    -> Timer Cell 12 and 13, IRQ 25
Deadline of TTTask1 at 950
    -> Timer Cell 14 and 15, IRQ 25
Activation of TTTask2 at 200
    -> Timer Cell 40 and 41, IRQ 32
Deadline of TTTask2 at 350
    -> Timer Cell 44 and 45, IRQ 33

Control Cell for
Timer Cells 08-15 and 40-47
    -> Control Cell GTC08
```

Figure 5.4: Example SLOTH ON TIME application configuration and example mapping to timer cells. The example configuration on the left is for the SLOTH ON TIME application introduced in Table 4.5 and Figure 4.8. The SLOTH ON TIME analysis and generation tool maps the activations and deadlines to available timer cells and corresponding IRQ sources on the target hardware platform; an example mapping for the Infineon TriCore TC1796 is depicted on the right.

task has completed its execution within time), restores the full context of the previously preempted task, and returns from the interrupt to continue execution of the preempted task.

only cell 13 generates the IRQ, which is why only cell 13 has its request enabled and disabled by the prologue and epilogue, respectively.

```
void handlerTTTask1(void) __attribute__((interrupt)) {
  /* prologue */
  savePreemptedContext();
  cell13.requestEnable = 1; /* deadline at 450 */
  cell15.requestEnable = 1; /* deadline at 950 */
  setCPUPriority(executionPriority);
  _enable(); /* enable interrupts */

  userFunctionTTTask1();

  /* epilogue */
  _disable(); /* disable interrupts */
  cell13.requestEnable = 0; /* deadline at 450 */
  cell15.requestEnable = 0; /* deadline at 950 */
  restorePreemptedContext();
  /* return from interrupt, re-enabling interrupts */
}
```

Figure 5.5: Example prologue and epilogue wrapper generated by SLOTH ON TIME. The example code is the generation result for `TTTask1` from the application configured in Figure 5.4.

5.2 Build Process and Compilation Structure

Since SLOTH relies on the hardware platform to perform as many operating system tasks as possible for it, its system calls are very small—both in terms of lines of code and in terms of compiled code size (see also evaluation in Section 6.2.1 and Section 6.4). That is why system call inlining is especially attractive for SLOTH applications and why the SLOTH file and compilation structure aims at facilitating inlining by the compiler.[2] Inlining is particularly worthwhile for short functions—which SLOTH system call implementations generally are—since the code size and performance overhead for the function call including saving and restoring registers can otherwise dominate the overhead of the function implementation itself. The drawbacks of inlining are a possible increase in code size depending on the number of function calls and the size of the function itself and a decreased cache locality; however, the latter is not applicable to many microcontroller platforms, which do not possess a cache. Since inlining is a trade-off decision, the application programmer can decide whether to enable it or not in the SLOTH system configuration.

The whole target binary, consisting of the application and the SLOTH operating system, is compiled from a single compilation unit to allow for whole-program optimization; only the platform-specific start-up code, which is usually contained in a single function anyway, is compiled and linked to the SLOTH binary in a separate step, since many platforms and compilers provide the start-up code in assembly, which has to be assembled by a separate program. The main compila-

[2] Note that system calls in operating systems of the class that SLOTH targets are usually function calls and not full traps; most corresponding microcontroller platforms do not even have a trap mechanism at their disposal.

tion unit, however, is a single C file: the `app.c` file provided by the application, containing the application logic in its tasks and other control flows. At the very top, this application file includes a single SLOTH operating system header file, `os.h`. This SLOTH master header in turn includes headers for different parts of the system before including the inline *implementations* of the operating system functions declared in the headers. The operating system function implementations are located in `*.inl` files, clearly separating them from the `*.h` declaration headers. Thus, the compilation unit includes both application-specific, generated SLOTH parts (see middle blue part in Figure 5.2) and application-independent, static SLOTH operating system parts (see lower red part in Figure 5.2).

The SLOTH build system then executes the C cross-compiler only once for a given application, compiling its `app.c` file with the inline SLOTH operating system implementation. This way, besides considering all SLOTH system calls for inlining in the application code due to their brevity, the compiler can perform comprehensive whole-program optimizations to optimize the binary for performance, code size, and data size, which is especially important in the target domain of small embedded systems.

5.3 Implementation File Structure

The clear separation of platform-independent and platform-specific parts of the operating system as well as application-independent and application-specific generated parts as introduced in Figure 5.2 is also reflected by the SLOTH operating system directories and files, which are located in the `system/` directory of the SLOTH root directory:

Application-independent, architecture-independent parts
: These parts are static operating system parts and can be found in files located directly in `system/`. They include modules for OSEK-inspired system calls (`osek.[h,inl]`), AUTOSAR-inspired system calls (`autosar.[h,inl]`), and other RTOS system calls (`rtos.[h,inl]`). Additional, general operating system services can be found in `kernel.[h,inl]`, and the architecture-independent interface to architecture-specific implementations is located in `system.h`.

Application-specific, architecture-independent parts
: These parts can be found in the `generate/` subdirectory of the `system/` directory, since they need to be generated depending on the application configuration. That is why those parts are Perl template files that generate C files after evaluating the configuration.

Application-independent, architecture-specific parts
: These parts can be found in the `$ARCH/` subdirectories of the `system/` directory, named after a short platform name. They include a header for

platform-specific functions (named `$ARCH_*()`), the platform-specific system implementation in `system.inl` (which implements both the platform-independent and the platform-specific header), the linker file, and the start-up code.

Application-specific, architecture-specific parts
These parts can be found in the `$ARCH/generate/` subdirectory of the `system/` directory. They constitute the flexible part of the SLOTH HAL (see also Section 3.3.2) and include templates for C headers and implementation files that need to be generated depending on the application configuration for the specific hardware platform.

The separation of architecture-specific and architecture-independent files in an operating system implementation has a long tradition to ease portability. In addition, SLOTH distinguishes between application-specific code parts, which have to be generated according to the configuration, and application-independent code parts, which are supplied statically by SLOTH. During compilation, these four parts are combined to constitute the SLOTH operating system with its tailored system calls and universal control flow abstraction adapted to the application (see Figure 5.2).

5.4 Hardware Platform Requirements for an Ideal SLOTH Implementation

Since the SLOTH operating system design is hardware-centric, the approach has requirements on the hardware platform to make an ideal implementation feasible; if parts of those requirements are not fulfilled, the design adapts using software-based emulation where necessary. Depending on the type of operating system and its support for different operating system abstractions (basic tasks, extended tasks, time-triggered tasks), only a subset of the stated requirements need to be fulfilled for an ideal implementation: SLOTH only needs Requirements 1 through 3, only SLEEPY SLOTH systems additionally need Requirement 4, and only SLOTH ON TIME systems need Requirements 5 and 6.

1. **Interrupt priorities**
   The hardware interrupt system shall offer as many different interrupt priorities as there are tasks and ISRs in the configured application. This is necessary as both ISR and task priorities are mapped to dedicated interrupt priorities by SLOTH.

2. **Interrupt triggering**
   The hardware interrupt system shall support manual, software-based triggering of interrupts through a special instruction or through the modification of corresponding hardware registers. This is necessary to implement synchronous task activation and chaining in SLOTH.

3. **CPU priority raising**
   The hardware platform shall support temporarily raising the CPU priority and lowering it to the original priority again. This is necessary to support the resource abstraction and operating system synchronization in SLOTH.

4. **IRQ source masking**
   The hardware interrupt system shall support masking and unmasking of individual IRQ sources. This is necessary to implement blocking and unblocking of extended tasks in SLEEPY SLOTH systems.

5. **Timer cells**
   Ideally, the hardware timer system shall offer as many timer cells as there are task activations and deadlines in the configured application dispatcher round. This is necessary to implement time-triggered actions in SLOTH ON TIME; however, if fewer timer cells are available, cells can be multiplexed at the cost of slightly increased memory overhead and run time overhead.

6. **Timer cell IRQs**
   The hardware timer system shall allow for each timer cell to trigger a separate IRQ. This is necessary to directly attach task functions to time-triggered activations in SLOTH ON TIME; however, if several timer cells share an IRQ, they can still be leveraged to activate the *same* task at different points within a dispatcher round.

Some hardware platforms fulfill all of these requirements natively (such as the Infineon TriCore, the reference platform for the SLOTH operating systems), whereas others have external chips that provide the corresponding functionality (such as the APIC interrupt controller chip present on all modern Intel x86 systems); the subset of microcontrollers that fulfills all of the requirements is called the SLOTH domain of microcontrollers, which enables an ideal SLOTH operating system implementation (compare [CMMS79]). Concerning the availability of timer cells and IRQs for time-triggered SLOTH ON TIME systems, many modern microcontrollers, especially those that are used in control systems, offer plenty of configurable timers—like the Freescale MPC55xx and MPC56xx embedded PowerPC families and the Infineon TriCore TC1796. Table 5.1 shows the hardware platforms that SLOTH implementations are currently available for, together with the implemented feature set per platform.

5.5 Reference Implementation on the Infineon TriCore TC1796

The Infineon TriCore microcontroller [Infc] is an architecture that is widely used in the automotive industry; the TC1796 derivative [Infb] serves as a reference platform for the implementation of the SLOTH design. The TriCore is a 32-bit

Hardware platform	Implemented features
Infineon TriCore TC1796	Reference platform with full implementation of all SLOTH, SLEEPY SLOTH, and SLOTH ON TIME features.
Atmel SAM3U and ST STM32 (both ARM Cortex-M3)	Full SLOTH and SLEEPY SLOTH implementation.
Infineon AURIX TC277T	Full SLOTH and SLEEPY SLOTH implementation.
Intel x86	Full SLOTH implementation.
Freescale MPC5602P (Embedded PowerPC)	Full SLOTH implementation.
Freescale Kinetis KL0 (ARM Cortex-M0+)	Full SLOTH implementation.

Table 5.1: Hardware platforms currently supported by the SLOTH operating system. SLOTH is currently implemented on six different architectures in different operating system feature sets.

microcontroller and features a RISC (reduced-instruction-set computing) load–store architecture and a Harvard memory model; its interrupt system has 256 priority levels and the TC1796 chip has about as many interrupt sources with memory-mapped registers. The platform fulfills all of the requirements stated in Section 5.4 for an ideal SLOTH implementation; this section describes the peculiarities of the platform as relevant for the SLOTH operating systems implemented on top of it.

5.5.1 TriCore Interrupt System

As an abstraction for interrupt sources, the TriCore microcontroller features service request nodes (SRNs), which encapsulate control and status fields for the interrupt source; the registers are mapped to memory. Most SRNs are connected to a periphery device; additionally, SRNs for software access only are available. SLOTH uses the latter SRNs for tasks that are only activated within software, and it uses those connected to the timer array for tasks that can be activated by alarms and for time-triggered tasks (see also Section 5.5.4).

Interrupt Control Registers

The memory-mapped registers per SRN used by SLOTH include:

- `SRPN` (service request priority number): IRQ priority in the range 0–255. Used by SLOTH as a static task priority in the initialization (see Requirement 1 in Section 5.4).
- `SETR` (service request set bit): triggers the corresponding IRQ. Used by SLOTH to activate a task and to auto-start a task during start-up (see Requirement 2 in Section 5.4).

- `SRE` (service request enable control): enables the corresponding IRQ. Used by SLOTH to enable tasks in the initialization and by SLEEPY SLOTH to block and unblock tasks (see Requirement 4 in Section 5.4).
- `SRR` (service request flag): queries whether an IRQ is pending. Used by SLOTH for hardware–software synchronization in the interrupt arbitration phase (see description later in this section).

Additional, global interrupt registers used by SLOTH include:

- `CCPN` (current CPU priority number): current IRQ priority. Used by SLOTH to temporarily raise the IRQ priority for operating system synchronization purposes[3] and to implement application resources (see Requirement 3 in Section 5.4).
- `IE` (global interrupt enable bit): enables the global recognition of IRQs. Used by SLOTH for synchronization purposes *with the IRQ arbitration system* via the instructions `disable` and `enable`.

Interrupt Arbitration Synchronization

The TriCore interrupt arbitration system features latencies that have to be respected by the operating system for semantical correctness. The TriCore's SRNs are all connected to a dedicated interrupt arbitration unit via a special bus for exchanging priority information in order to find a precedence among the pending interrupts. This arbitration process takes a defined number of system bus cycles; this time itself depends on the system clock frequency and the priority range of the SRNs actually competing in the arbitration and is specified by Infineon in an application note [Infa]. Thus, the fewer tasks and ISRs are configured in a SLOTH system, the fewer arbitration cycles are needed to prioritize the potentially concurrent requests. On the TriCore, the number of needed cycles grows logarithmically to the number of configured priorities—1 cycle for up to 4 priorities, 2 cycles for up to 16 priorities, 3 cycles for up to 64 priorities, and 4 cycles for up to 256 priorities. On a different hardware platform, this arbitration might be implemented in parallel in the hardware so that the search overhead is constant.

Since the SLOTH operating system uses hardware mechanisms to implement software features like task scheduling and dispatching, special attention has therefore to be paid to synchronize hardware and operating system software. For instance, synchronous task activation is performed by requesting the corresponding interrupt using the appropriate SRN. Basically, this is compiled to a single store instruction to the memory-mapped `SETR` register. However, it takes a defined amount of time [Infa] until the interrupt request is propagated to the

[3] Note that the TriCore architecture does not feature a compare-and-swap or similar instruction to consider non-blocking synchronization for operating system synchronization in the domain of resource-constrained embedded real-time systems (see discussion in Section 2.2.1).

CPU, depending on the current state of the arbitration system. Since an activation of a higher-priority task is supposed to happen *synchronously* in a preemptive system, this activation has to be synchronized. As an example, consider a preemptive application featuring a `Task1` with priority 1 and a `Task3` with priority 3. Whenever `Task1` executes `ActivateTask(Task3)`, it expects the operating system to dispatch and execute `Task3` *immediately* and *before* executing any `Task1` instructions following the `ActivateTask()` call. The application *relies* on these synchronous system call execution semantics.

SLOTH synchronizes the `ActivateTask()` system call as shown in Figure 5.6(a). It first disables all task interrupts by locking the operating system before the interrupt triggering via `tc1796Setr()`, and by then reading back the `SRR` request bit in order to synchronize the hardware and software as specified by Infineon [Infa]. The interrupt request will then be considered by the arbitration system, which takes a defined amount of time [Infa]. Thus, `nop` instructions are inserted to accommodate for the worst-case latency, which arises if an arbitration round has just begun without the new request, so an additional arbitration round has to be waited for. The number of `nop` instructions to be inserted is bounded and calculated statically by the static analyzer (see Section 5.1.2), depending on the number of arbitration rounds and the number of cycles per arbitration round as demanded by the application configuration (i.e., the number of tasks and the system frequency); the number is encapsulated in the routine `tc1796NopsForOneArb()`. The subsequent code to enable all task interrupts by unlocking the operating system is then the defined synchronous point of preemption by the activated task if it has a higher priority (see Figure 5.6(a)). After that, the new interrupt is guaranteed to have been respected by the arbitration. Without executing the `nop` instructions, the task preemption interrupt could occur at any point *after* the actual system call in the application task.

This kind of implementation also eliminates any possible jitter for preemptive task activations. Without locking the task interrupts during the execution of the `nop` instructions, the task preemption interrupt could occur at any point in the `nop` sequence, introducing jitter to both the task preemption and the return to the preempted task, which would then still execute some `nop` instructions.

An alternative implementation trades *increased* jitter for better average-case performance. If this behavior is preferred and configured by the application, SLOTH executes the `nop` instructions *after* re-enabling the interrupts. This way, in the average case, the preemption will take place faster, since the number of `nop` instructions is calculated for the worst case but is not needed in most of the cases. On the other hand, the system will exhibit increased jitter both for task activations and task terminations, since the number of `nop` instructions executed before and after the preemption will vary depending on the current state of the arbitration system.

SLOTH can also offer an asynchronous version of the system call—`ActivateTaskAsync()`—if the synchronous preemption semantics is not needed by the application. The asynchronous system call does not guarantee immediate pre-

```
INLINE void ActivateTask(TaskType id) {
  lockKernel(); /* disable all task interrupts */
  tc1796Setr(id); /* set service request flag */
  tc1796Srr(id); /* read back to sync hardware and software */
  /* worst case: wait for 2 arbitrations */
  tc1796NopsForOneArb(); tc1796NopsForOneArb();
  unlockKernel(); /* enable all task interrupts; defined preemption point */
}
```

(a) TC1796 implementation and arbitration synchronization of the `ActivateTask()` system call.

```
INLINE void WaitEvent(EventMaskType mask) {
  lockKernel(); /* disable all task interrupts */
  if ((eventMask[currentTask] & mask) == 0) {
    /* none of the events has already been set */
    eventsWaitingFor[currentTask] = mask;
    /* block task */
    archDisableIRQSource(currentTask);
    archSetCPUPrio(0);
    /* worst case: wait for 2 arbitrations */
    tc1796NopsForOneArb(); tc1796NopsForOneArb();
  }
  unlockKernel(); /* enable all task interrupts; defined preemption point */
}
```

(b) TC1796 implementation and arbitration synchronization of the `WaitEvent()` system call.

```
INLINE void SetEvent(TaskType id, EventMaskType mask) {
  lockKernel(); /* guard non-atomic mask modification and check */
  eventMask[id] |= mask;
  if ((eventMask[id] & eventsWaitingFor[id]) != 0) {
    /* at least one of the events that the task has been waiting for is set */
    eventsWaitingFor[id] = 0;
    unlockKernel();
    /* unblock task */
    lockKernel(); /* disable all task interrupts */
    archEnableIRQSource(id);
    /* worst case: wait for 2 arbitrations */
    tc1796NopsForOneArb(); tc1796NopsForOneArb();
    unlockKernel(); /* enable all task interrupts; defined preemption point */
  }
}
```

(c) TC1796 implementation and arbitration synchronization of the `SetEvent()` system call.

Figure 5.6: Implementation sketch of selected system calls that need arbitration synchronization on the Infineon TriCore TC1796. Arbitration is needed when (a) activating a task, (b) blocking a task, and (c) unblocking a task, amongst others.

emption, so it has to be particularly considered by the application. For instance, an application task can safely use the asynchronous system call and benefit from a lower system call latency without risking to compromise data integrity if it knows beforehand that it does not share data with the preemptively activated task. The same applies to a task that activates a lower-priority task and keeps executing

for a certain amount of time—thereby avoiding potential race situations. The implementation of the asynchronous system call simply pends the corresponding interrupt without synchronizing with the arbitration system afterwards, refraining from giving any preemption guarantees to the caller. This way, the application is provided with kind of a "delay slot", which it can fill with appropriate functionality.

The same applies to the chaining of another task: The executing task relies on the chained task to be executed immediately after it terminates if that new task has the highest priority in the system at that point. In this case, as described in Section 4.2.1, interrupts are also disabled before the activation in order to prevent the new task from running until the previous one has terminated.

Even when a *lower-priority* task is activated, this situation may require synchronization. Consider, for instance, that directly after the (non-synchronized) activation of a lower-priority task, the priority level is lowered by terminating the running task. This has to be the defined point for the context switch, and not when the interrupt actually occurs at the CPU a couple of cycles later. If the activation is not synchronized, a lowest-priority task may execute for a few cycles *after* the termination of the high-priority task and *before* the interrupt controller dispatches the activated task—which is a violation of the semantics expected by the application programmer, who relies on *synchronous* task activation in the corresponding system call. Hence, *every* task activation is synchronized with `nop` timing as described above, independent of its priority.

Similar situations need to be respected in the implementation of `WaitEvent()` and `SetEvent()` when blocking and unblocking a task, respectively (see Figure 5.6(b) and Figure 5.6(c)). In both cases, the hardware interrupt priority state and pending priorities are altered (disabling an IRQ source plus setting the CPU priority to zero, and re-enabling an IRQ source, respectively), and a high-priority task can be the direct successor, so interrupt arbitration synchronization using the appropriate number of `nop` instructions is also needed. Again, the defined point for preemption after blocking a task or unblocking it is the point after unlocking the operating system at the end of the respective system call.

All of the cases described in this section where the operating system needs to be locked temporarily for synchronization purposes only locks it for a *short* and *bounded* amount of time. That way, that time can be accounted for during the schedulability and latency analysis of the whole real-time system. The number of introduced `nop` instructions varies between 8 and 22, depending on the configuration, and is effectively time when the CPU cannot do useful work (although the interrupt system is performing the priority arbitration during that time), making up for most of the hardware-induced costs of the corresponding system calls. However, this is a small price to pay compared to the overhead of a traditional, software-based scheduler implementation (see also evaluation in Chapter 6).

Additionally, the `lockKernel()` directive is subject to a trade-off decision that can be configured by the application programmer. One alternative is to map the directive to a `disable` instruction, which disables *all* interrupts. This keeps the

code size small (1 instruction) and the system calls fast (1 CPU cycle), but it raises the worst-case latency for unrelated category-1 interrupts and a time-triggered system running on top. The other alternative is to map the `lockKernel()` directive to raising the CPU priority to the maximum priority of all tasks and category-2 ISRs in the system to reach the same synchronization behavior. This way, higher-priority category-1 interrupts are unaffected in their latencies, but the code size (4 instructions) and the system call latencies (4 CPU cycles) will rise.

5.5.2 TriCore Context Switches

The TriCore architecture has both a regular stack to allocate function-local data and an additional call stack. This call stack is used by the hardware to automatically save certain register sets depending on the causing event (e.g., a function call or an interrupt). The hardware architecture implements this call stack through linked lists of so-called context save areas (CSAs), which hold 16 word-sized registers (4 bytes each, 64 bytes total). One CSA can either hold a so-called upper context (the non-volatile set of computation registers) or a lower context (the volatile set of computation registers); both include the `A11/RA` register (to store the return address) and a link word to refer to the next CSA in the linked list. Upon interrupt entry, the hardware automatically saves the upper context of the preempted task, so SLOTH only needs to save the lower context manually using the special `svlcx` instruction.

At boot time, the start-up code reserves part of the RAM for CSA usage and links the CSAs in a free context list, pointed to by the `FCX` register. During program execution, the `PCX` register points to the previous context list, which includes all contexts that have been saved in the current control flow. Additionally, there is an `LCX` register that points to one of the last CSAs in the free context list; if that CSA is used by the hardware, an impeding free CSA list depletion is signaled by a special trap.

Regular SLOTH with run-to-completion tasks runs on a single stack and a single CSA stack; since the system has a stack-based execution pattern with strictly nested function calls and interrupt preemption, nothing special has to be considered in that case after correctly initializing the CSA list. The same holds true for generalized time-triggered SLOTH ON TIME systems, which also exhibit a stack-based execution pattern. Such systems are also executed on a common data stack and a common CSA stack.

In SLEEPY SLOTH systems, however, extended tasks can be blocked, so they need to run on stacks *and* CSA stacks of their own. That is why SLEEPY SLOTH maintains both a stack pointer and a `PCXI` pointer (pointing to the CSA that has most recently been used by the task, together with information about the preempted interrupt priority) for each task. The data stack pointer is implicitly saved in the last upper context that has been saved before switching tasks, so SLEEPY SLOTH only has to save the task's `PCXI` pointer to an operating system array in memory. To save space in RAM, SLEEPY SLOTH uses a common data stack

and CSA stack for all *basic* tasks, which among themselves exhibit a stack-based pattern and can therefore run on the same stacks. If SLEEPY SLOTH initializes a context for a task that is about to run for the first time, it sets the data stack pointer to the top of the task stack and sets the `PCXI` register to zero so that the task starts building up a CSA stack of its own.

5.5.3 TriCore Optimizations

The TriCore microcontroller architecture features some peculiarities that are used by the SLOTH operating systems to optimize its performance and memory footprint.

For one, the TriCore possesses four global registers, which are not used by the compiler and are therefore never automatically saved and restored. SLOTH uses two of those registers to contain heavily accessed operating system data fields: the current task ID, which is often used as an index by the operating system, and the pointer to the task ID stack, which is used upon dispatch and termination of a task. Read and write commands by the operating system to those data fields therefore consist of a single read or write instruction from or to the corresponding global register.

Furthermore, the interrupt vector table of the TriCore architecture is special in the sense that it does not include the address of the corresponding handler function but the beginning of the handler function itself. Thus, upon receiving an interrupt, the CPU directly executes the first instruction of the interrupt handler without having to look up an address in a table. Each interrupt slot is 32 bytes, so any interrupt handler that fits into those boundaries can be directly accommodated in the vector table without a single jump and look-up. The task wrapper code of the original SLOTH operating system is optimized so that its prologue part fits exactly into such an interrupt slot; the last instruction is a jump directly to the user task function (see Figure 5.7). This way, SLOTH minimizes the basic task latency. Additionally, if the application features a very small ISR—which only sets a single value, for instance—SLOTH can directly embed this application code into the interrupt slot if the resulting total size is less than 32 bytes.

An additional optimization is concerned with the ceiling priority of resources. The SLOTH implementation uses C enumerations for the names of the resources as configured by the application programmer, and it directly assigns the ceiling priority as calculated by the generator to the corresponding resource name. This way, the resource system calls are directly passed the ceiling priorities as a parameter and do not have to look them up via an identifier. The system calls can therefore be kept smaller in code size and faster in their execution times.

5.5.4 TriCore Timer System

The TriCore TC1796 microcontroller features a sophisticated timer system called its general-purpose timer array (GPTA), which the SLOTH ON TIME implementa-

```
<handlerTask42>:
    mov %d0, 0x0b80
    mtcr $psw, %d0        // configure PSW to allow global register writes
    isync                 // make PSW changes visible to the following instructions
    st.a [+%a9], -4, %a8  // save preempted task ID (in A8) to ID stack (pointer in A9)
    mov.a %a8, 42         // set current task ID
    svlcx                 // save lower context
    enable                // enable interrupts
    j functionTask42      // jump to user task function
```

Figure 5.7: Example SLOTH interrupt vector table entry on the Infineon TriCore platform. The vector table entry is generated for a SLOTH task `Task42` with ID 42; the instructions fit exactly in the 32 bytes of an interrupt slot.

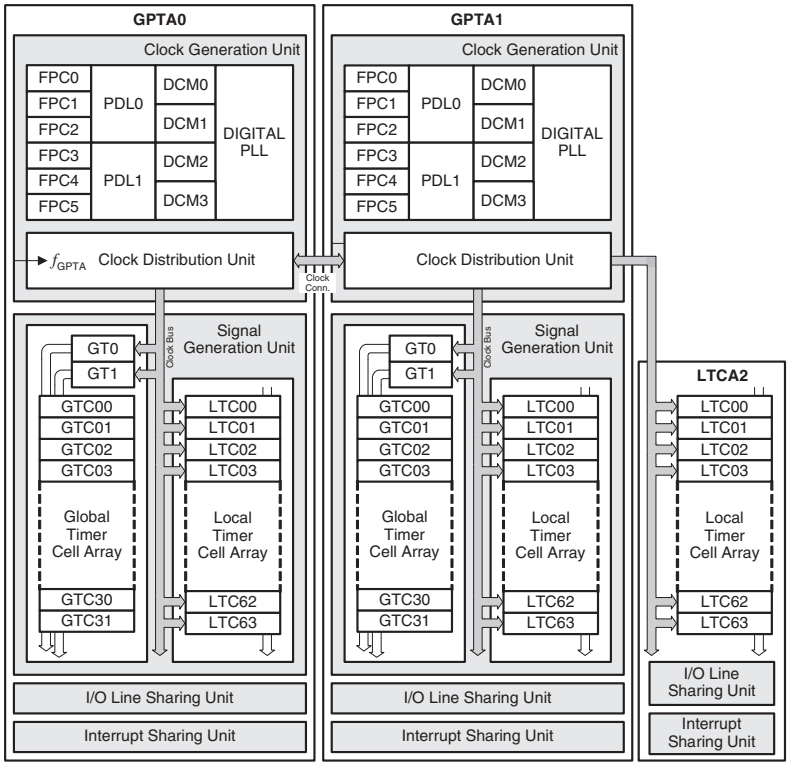


Figure 5.8: Overview of the GPTA timer module on the TC1796 microcontroller. The GPTA timer features 256 timer cells connected to 92 interrupt sources. (Graphics taken from [Infb].)

tion uses to provide an ideal implementation of generalized time-triggered task operation (see Section 5.4), and which contains 256 timer cells in total. An overview of the GPTA module is shown in Figure 5.8.

The whole timer system contains three GPTA modules: GPTA0, GPTA1, and LTCA2. The modules contain subunits to generate and distribute different clocks

to the cells, the actual timer cells themselves, and two sharing units, which distribute the signals among the cells and the interrupt unit. The GPTA is connected to 92 interrupt SRNs, which can be used by SLOTH ON TIME to activate tasks and monitor deadlines. Among its timer cells, the GPTA distinguishes between so-called global timer cells (GTCs) and local timer cells (LTCs), which exhibit different operation modes. SLOTH ON TIME uses LTCs for its actual operation and GTCs for simultaneous starting and stopping of a dispatcher round.

Due to the way its modes and interconnections work, two adjacent LTCs have to be used on the TC1796 for one operating point (e.g., an activation point). The first cell is set to operate in reset timer mode, and its internal register is used as the corresponding counter. The second cell is put into compare mode; its internal register is used as the corresponding compare value. On a match compared to the counter value provided by the first cell, the second cell triggers an interrupt and prompts the first cell to reset its counter value. The first cell will then re-start counting upwards, providing the desired cyclic behavior. By clearing the request enable bit of the second cell, the interrupt generation can be suspended without affecting the cyclic operation of the cells in the background; this feature is used by the optimized SLOTH ON TIME deadline monitoring to avoid unnecessary interrupts (see Section 4.4.4). Additionally, the TC1796 timer cells can be run in one-shot mode, which is used by SLOTH ON TIME to implement non-repeating schedule tables. When the corresponding bit is set in all cells for a schedule table, the cells are automatically de-activated after one round has elapsed—without having to intervene using software.

The GPTA can also be configured to link cells hierarchically, which is used by SLOTH ON TIME to implement control cells to start all cells in a dispatcher round simultaneously (see Section 4.4.3). To achieve this, the I/O line sharing unit is configured to connect the output line of a GTC (used as the control cell) to the input lines of the LTCs used for a dispatcher table. The GTC can then be used as a global switch to start or stop a complete dispatcher round in an atomic manner.

5.5.5 TriCore Operating System Synchronization

Since most of the operating system state like its ready queue is moved to the hardware by SLOTH, its system synchronization becomes very manageable. The software operating system state, which might need explicit synchronization, is restricted to three parts: the suspend counters for interrupt and OS interrupt suspension, the resource stack with its stack pointer, and the event masks of received and waited-for events in SLEEPY SLOTH systems. This section shows how the SLOTH TriCore implementation can keep explicit synchronization in its system calls to a minimum. As discussed in Section 5.5.1, the locking of the operating system can be configured to either disable all interrupts or to raise the priority to the maximum task and category-2 ISR priority, which is subject to a trade-off decision. The TriCore architecture does not feature a compare-and-swap instruction

```
<TerminateTask>:
    rslcx                 // restore lower context
    ld.a %a8, [%a9+], +4  // load current task ID from task ID stack
    rfe                   // return from interrupt, implicitly restoring upper context
```

Figure 5.9: Assembly code for terminating a task on the Infineon TriCore platform. The termination of a task effectively yields three machine instructions, providing for a very low overhead.

or similar instructions to be used for non-blocking synchronization constructs for the operating system (see also discussion in Section 2.2.1).

SLOTH: `ActivateTask()` and `ChainTask()`

Both system calls lock the operating system at the beginning of their implementation and re-enable them at the end (see implementation of `ActivateTask()` in Figure 5.6(a)). However, this is not due to *operating system* synchronization reasons but for reasons related to *interrupt arbitration* (see Section 5.5.1). This is unavoidable in order to synchronize the operating system software with the interrupt hardware to have a defined point for preemption.

SLOTH: `TerminateTask()`

The termination of a task does not need explicit operating system synchronization since it is interruptible at any point (see Figure 5.9). Depending on whether the `TerminateTask()` system call is preempted before or after restoring the current task ID from the task ID stack, the old or new task is officially running; this fact does not matter to the preempting task. In any case, the CSA context stack is restored to the state before the preemption, so the preempted task is not affected by it.

SLOTH: `SuspendAllInterrupts()` and `ResumeAllInterrupts()`

At the beginning of the suspend system call, the operating system disables all interrupts to protect the subsequent alteration of the suspend counter needed to keep track of the nesting level (see Figure 5.10). However, it is also the *functionality* of the system call to disable all interrupts; thus, the `disable` instruction is needed in any case, independent of the synchronization mechanism used for the system call *itself*. By using a suspend counter instead of pushing the previous interrupt state onto a stack, the memory demand for the necessary stack can be saved.

Invoking the resume system call is only allowed after having invoked the suspend call, which can be ensured using static-analysis tools; thus, the system call is always called with interrupts disabled (see Figure 5.10). Hence, the alteration

```
INLINE void SuspendAllInterrupts() {
  _disable(); /* not interruptible after that */
  suspendAllCounter++; /* counter is guarded */
}

INLINE void ResumeAllInterrupts() {
  suspendAllCounter--; /* not interruptible here since interrupts are already suspended */
  if (suspendAllCounter == 0) _enable(); /* counter is guarded */
}
```

Figure 5.10: Code for suspending and resuming interrupts on the Infineon TriCore platform. The suspension counter is implicitly guarded against concurrent access by design.

and the test of the suspend counter are automatically protected against preemption.

SLOTH: `SuspendOSInterrupts()` and `ResumeOSInterrupts()`

Suspending the operating system interrupts works similar to suspending all interrupts, except that the CPU priority level is raised to the maximum of all task and category-2 ISR priorities instead of disabling interrupts completely. The corresponding system calls are synchronized similarly to protect the corresponding suspend counter; again, the raising of the priority level is also part of the system call functionality.

SLOTH: `GetResource()` and `ReleaseResource()`

Since resource acquisitions can be nested, the operating system implementation has to keep track of the previous execution priorities on a resource stack with a corresponding resource stack pointer (see Figure 5.11). This stack access has to be synchronized since the system call can be preempted by a higher-priority task; the SLOTH implementation therefore deploys a clever sequence of operations.

If the stack pointer is increased *before* pushing the previous priority, the slot on the stack is effectively reserved and the system call is protected against corruption by preempting tasks (see Figure 5.11(a)). If the read–modify–write cycle of the increase operation itself is interrupted, the exact same state of the stack pointer as before the interruption will be restored due to the stack-based preemption pattern of the task execution. The same holds true for the `ReleaseResource()` system call (see Figure 5.11(b)): If the access to the stack itself or the subsequent access to the stack pointer is interrupted, the preempting task will restore the exact same state as before the interruption before terminating itself.

Since the priority level must not decrease when first acquiring a high-priority resource and then acquiring a low-priority resource, a calculation of the new priority level is needed (see Figure 5.11(a)). Since the implementation caches

```
INLINE void GetResource(ResourceType id) {
  TaskType curPrio = archGetCPUPrio(); /* atomic read of current priority */
  resourceStackPointer++; /* reserves a slot in the stack */
  resourceStack[resourceStackPointer] = curPrio; /* access to reserved slot */
  /* resource ceiling priority is stored in ID itself */
  TaskType newPrio = (id > curPrio) ? id : curPrio; /* level must not decrease */
  archSetCPUPrio(newPrio);
}
```

(a) TC1796 implementation and implicit synchronization of the `GetResource()` system call.

```
INLINE void ReleaseResource(ResourceType id) {
  TaskType newPrio = resourceStack[resourceStackPointer]; /* get prio from stack */
  resourceStackPointer--;
  archSetCPUPrio(newPrio);
}
```

(b) TC1796 implementation and implicit synchronization of the `ReleaseResource()` system call.

Figure 5.11: Code for system calls related to resource management on the Infineon TriCore platform. The resource stack and its pointer are implicitly guarded against concurrent access by design when (a) acquiring a resource and (b) releasing a resource.

the current CPU priority at the beginning of the system call, alterations of the CPU priority by preempting tasks are non-critical; additionally, they will correctly restore it before returning.

SLEEPY SLOTH: `WaitEvent()` and `SetEvent()`

The SLEEPY SLOTH system calls to block (`WaitEvent()`) and unblock a task (`SetEvent()`) have been shown in Figure 5.6(b) and Figure 5.6(c), respectively. Besides the previously discussed synchronization with the arbitration system, they also need to be synchronized to avoid corruption of the event operating system data and to avoid lost-wake-up problems.

A lost-wake-up situation might occur if the blocking task is interrupted by the unblocking task *after* it has checked its current event mask but *before* actually blocking. In that case, the blocking task would lose the wake-up signal and would potentially be blocked indefinitely. Thus, `WaitEvent()` locks the operating system before checking the event mask. Interruptions of a blocking task by another task that blocks—and therefore concurrently accesses event operating system data—can therefore not occur either.

An interruption of `SetEvent()` by another `SetEvent()` invocation can cause a lost update in the event mask due to the necessary read–modify–write cycle; thus, the mask modification has to be guarded (see Figure 5.6(c)). Additionally, a possible interruption after the mask check but before re-setting the waited-for events mask will lead to a double unblocking; thus, this situation is also protected against.

An interruption of `SetEvent()` by `WaitEvent()` is not critical. If the interruption occurs after the mask modification, the interrupting task will block and will then be unblocked by the interrupted task. An interruption after a positive check in `SetEvent()` cannot occur, since the positive check means that the corresponding task is already blocked, so it cannot interrupt the current task.

5.6 Implementation on the Atmel SAM3U and Platform Differences

As noted in Section 5.4, SLOTH also runs on other microcontroller platforms besides the reference implementation for the Infineon TriCore TC1796 (see Table 5.1). All of those platforms fulfill the stated requirements for ideal implementations of the hardware-centric SLOTH design; the implementations, however, differ slightly due to platform peculiarities. As an example, this section lists those differences between the implementation on the Atmel SAM3U, an ARM-Cortex-M3-based microcontroller, and the TriCore reference implementation.

The main difference in the SAM3U SLOTH implementation results from the fact that the platform calculates its current CPU priority based on several factors; it does not feature an explicit current CPU priority field that can be manipulated in software as on the TC1796. For one, the SAM3U tracks the currently active interrupt handlers; the current priority is the one of the highest-priority active interrupt. This priority can be raised (but not lowered below the current IRQ priority) using the `BASEPRI` register; if that register has a non-zero value, all interrupts with a lower priority than the one specified by the value are deferred. Additionally, a set `PRIMASK` bit effectively disables all interrupts except the NMI and the hard fault handler, whereas a set `FAULTMASK` bit disables all interrupts except the NMI. On an interrupt return, the hardware keeps the `PRIMASK` bit the way it was, whereas the `FAULTMASK` bit is reset.

The SLOTH implementation uses the interrupt control registers on the SAM3U in the following ways:

1. **Resources in SLOTH**
   The resource implementation uses the `BASEPRI` register to raise the priority during the critical section according to the stack-based priority ceiling protocol (see Section 4.2.3). Since tasks can acquire multiple resources simultaneously, a comparison is needed to check if the priority level rises while taking the inner resource (see also Figure 5.11(a)). That comparison can be performed directly in hardware on the SAM3U using the `BASEPRI_MAX` register, which only alters the `BASEPRI` register if the level would rise. This hardware-based comparison saves 9 cycles in the running implementation by skipping the software-based comparison, lowering the overhead for the `GetResource()` system call from 29 to 20 clock cycles.

2. **Resources in SLEEPY SLOTH**
   Since the resource implementation on the TC1796 directly alters the CPU priority, a task preemption by an extended task in SLEEPY SLOTH will trigger a special resource IRQ upon termination (see Section 4.3.6). Such a dedicated resource IRQ is not necessary on the SAM3U, since the raised `BASEPRI` value remains unchanged upon preemption and therefore leads to the correctly raised priority level upon termination of the preempting task. This is a direct consequence of the separate priority alteration using the `BASEPRI` register instead of an explicit alteration of the current CPU priority.

3. **SLEEPY SLOTH extended task contexts**
   Due to the separation of the call stack and the data stack on the TC1796, both a data stack pointer and a CSA stack pointer have to be maintained; however, the whole context is contained in the task CSAs, so the CSA stack pointer holds the state of the task. On the SAM3U, the task context needs to be pushed on the data stack; the stack pointer then holds the task state and is saved in the SLOTH operating system context array.

4. **SLEEPY SLOTH extended task blocking**
   Due to the fact that the SAM3U cannot *lower* the CPU priority, the SLEEPY SLOTH mechanism to yield the CPU when blocking by lowering the priority to zero cannot be applied. Instead, the implementation has to internally terminate the task by returning from the interrupt to remove the interrupt from the active interrupts tracked by the SAM3U, effectively allowing the task with the next-highest priority to run. Since the blocking task needs to be continued upon unblocking, its context including the return address is saved to the stack and the stack pointer is stored in an operating system array before returning from the interrupt. The context is restored upon unblocking by the task prologue as described in Section 4.3.1.

5. **System-internal task chain synchronization**
   Chaining a task means activating it and terminating the currently running task in one atomic system call, so the operating system needs to synchronize it. On the TC1796, this is done by disabling interrupts during the critical section (see Section 5.5.1); the interrupts are implicitly re-enabled when returning from the interrupt at the end of the system call. On the SAM3U, the corresponding `BASEPRI` register is *not* reset upon a return from interrupt, so the SLOTH implementation uses the `FAULTMASK` register—which the hardware *does* reset upon a return from interrupt—to synchronize the system call.

6. **Synchronization of pending IRQs**
   The TC1796 implementation of a task activation needs an explicit synchronization with the interrupt arbitration system using a statically calculated

number of `nop` instructions (see Section 5.5.1). On the SAM3U platform, the instruction that pends an IRQ (corresponding to a task activation) is automatically synchronized by the hardware since the involved `ISPR` register is mapped into the system control address space. This address space is defined as strongly-ordered memory and in addition guarantees that side effects of any access that performs a context-altering operation take effect when the access completes [ARM10, p. A3-119]. The following `dsb` instruction (data synchronization barrier) guarantees the completion of this access to pend an IRQ. Flushing the instruction pipeline using an `isb` instruction (instruction synchronization barrier) is not necessary, since the possible branch to the interrupt handler is an exception entry, which itself flushes the pipeline.

7. **Alarm interrupt acknowledgment**
   The alarm implementation on the SAM3U uses the timer counters provided by the microcontroller platform. On the SAM3U, tasks that can be activated by alarms need slightly adapted task wrappers since they have to acknowledge the external timer interrupt. This adaptation can be accomplished within a single additional instruction; the mechanism also works when the task is activated synchronously using a system call.

8. **Interrupt priority space**
   In contrast to the TC1796, the SAM3U allocates higher priorities to lower priority numbers; priority number 255 is the lowest priority. Thus, the SLOTH system generator maps the logical priorities as specified by the application programmer to physical priorities in the reverse SAM3U priority space (see also Section 5.1.2).

5.7 Implementation Summary

The key of the SLOTH operating system is that it is tailored both to the application running above and to the hardware platform located below. In order to be able to ideally adapt to both of them, SLOTH comes with a comprehensive build framework. First, the framework analyzes the static configuration properties as specified by the application programmer and deduces and calculates information that is needed by the subsequent generation step; additionally, it runs verification checks on the configuration parameters. The system generation part of the framework then takes operating system template files and the configured and deduced information to provide application-specific operating system files, which, together with static operating system files, are compiled to the optimized SLOTH application–system binary. The SLOTH build process architecture and its file structure reflect and support that approach.

The hardware-centric nature of the SLOTH approach poses a couple of requirements on the underlying hardware platform for an ideal implementation;

proof-of-concept implementations for five different architectures are currently functional. The reference implementation on the Infineon TriCore is tightly integrated with its interrupt controller and timer array, and it makes use of the specialized context management provided by the platform in hardware. A comparison with a second implementation, for an ARM Cortex-M3 microcontroller, shows that SLOTH implementations are also feasible on platforms with different interrupt execution semantics and that implementations of the SLOTH design are similar in their hardware-based nature.

Evaluation 6

Sid, the SLOTH*, is drawing a* SLOTH *onto a stone with a piece of chalk.*

Diego, the saber-toothed tiger: What are you doing?
Sid: I'm putting SLOTHS on the map.
Manny, the mammoth: Why don't you make him more realistic and draw him lying down?
Diego: And make him rounder.

From the movie Ice Age*, 2002*

Since embedded real-time operating systems of a particular class are distinguished by their non-functional properties such as memory footprint, system call latencies, and maintainability, this chapter shows the results of the comprehensive evaluation of the SLOTH operating systems. Since the SLOTH design aims at making more use of existing hardware features than traditional operating systems, its software implementation is accordingly very concise if the hardware platform offers more sophisticated astractions. From a *functional* point of view, the SLOTH operating systems implement OSEK-like system calls in order to be able to compare them in an unbiased way to other implementations of those standards that do not follow the hardware-centric SLOTH approach.

As a proposed guideline, the OSEK OS specification [OSE05] lists implementation parameters to be provided by operating system implementations, which the following chapter takes into consideration as a basis for the evaluation. Besides limitations on the maximum number of supported operating system objects, this includes hardware resources used by the operating system and performance numbers, all of which are presented in this chapter:

1. RAM and ROM requirement for each of the operating system components (see Section 6.2)
2. Size for each linkable module (see Section 6.2)
3. Application-dependent RAM and ROM requirements for operating system data (see Section 6.2)

4. Execution context of the operating system (see Section 6.2)
5. Timer units reserved for the operating system (see Section 6.6)
6. Interrupts, traps, and other hardware resources occupied by the operating system (see Section 6.6)
7. Total execution time for each service (see Section 6.3)
8. Operating system start-up time without invoking hook routines (see Section 6.3)
9. Interrupt latency for ISRs of category 1 and 2 (see Section 6.3)
10. Task switching times for all types of switching (see Section 6.3)
11. Idle CPU overhead / base load of system without applications running (see Section 6.3)

In addition to the evaluation parameters suggested by the specification, the ability to certify an embedded system is influenced by the lines of source code (see Section 6.4); furthermore, the SLOTH operating systems are examined for their characteristics in situations of potential priority inversion, especially rate-monotonic priority inversion (see Section 6.5).

6.1 Evaluation Setting

The evaluation investigates the SLOTH operating systems in their implementations for the SLOTH reference platform, the 32-bit Infineon TriCore microcontroller in its TC1796 derivative, as introduced in Section 5.5. The SLOTH operating systems are compared to commercial TriCore implementations of the corresponding OSEK OS, OSEKtime OS, and AUTOSAR OS standards, all of which are used in standard cores of major automotive manufacturers[1].

Since both SLOTH and the commercial operating systems implement the same standards, the benchmark applications are written against the standard interface and are then compiled and run unaltered on all systems. This way, the results between the SLOTH operating systems and the commercial operating systems are directly comparable. The compiler in use is `tricore-gcc` by HighTec, with performance-oriented `-O3` optimizations enabled for all of the evaluation test cases (also for the memory footprint tests, which would perform better using size-oriented `-Os` optimizations in the trade-off). The version used for compiling the SLOTH systems is 4.6.1.3, the commercial systems are shipped with and can only be compiled with versions 4.5.1 (commercial OSEK and OSEKtime systems)

[1] Note that the exact names and versions of the commercial systems cannot be stated here for legal reasons, but they were disclosed to the peer reviewers of the accepted conference publications listed in Section 1.6.

Application	Configured operating system objects and features
Minimal BCC1 app	1 basic task
Maximum BCC1 app	8 basic tasks, 2 resources, 1 alarm, ISR support
Minimal ECC1 app	1 extended task
Maximum ECC1 app	8 basic tasks and 8 extended tasks with 8 events each, 10 resources, 1 alarm, ISR support
Minimal OSEKtime app	1 task activation
Maximum OSEKtime app	8 activations of 8 tasks, 8 deadlines
Minimal AUTOSAR app	1 schedule table with 1 extended task activation
Maximum AUTOSAR app	2 schedule tables with 16 activations of 8 basic tasks and 8 extended tasks with 8 events each, 10 resources, execution budgeting, ISR support

Table 6.1: Example evaluation applications and their configurations. The example applications are used for the evaluation of the SLOTH systems and the commercial operating systems in order to be able to compare the non-functional properties of the operating system implementations.

and 3.4.6 (commercial AUTOSAR system). For the evaluation, all systems were configured to the minimum feature set possible that supports the corresponding test case.

Where possible, the SLOTH characteristics and corresponding numbers are specified in a general and application-independent manner. However, since SLOTH systems *do* depend on the application configuration, eight test applications are used to show example evaluation numbers for a bandwidth of possible applications. For the four classes of an OSEK BCC1 system, an OSEK ECC1 system, an OSEKtime system, and a combined AUTOSAR-OS-like system, minimal and maximum configurations are specified as listed in Table 6.1. The configurations of the test applications are motivated by the corresponding standards and their specifications of the supported number of operating system objects.

Note that the stated microbenchmark results cannot always be directly added to yield results for more complex systems. In many cases, there are synergies resulting from the combination of several features, which are exploited by the SLOTH system generator or by the compiler. However, the microbenchmarks can be used to compare properties *between* different system implementations, which is the main purpose of this evaluation chapter.

6.2 Operating System Code and Data Size

Due to its hardware-centric design, the software part of the SLOTH operating system that an application has to deploy can be very small on a sophisticated microcontroller like the TriCore, since such controllers take over duties that the

System call implementation	Size in B	Dependencies
`ActivateTask()`	50 – 62	Arbitration rounds
`TerminateTask()`	16 – 92	Arbitration rounds, event support
`ChainTask()`	64 – 140	Arbitration rounds, event support
`Schedule()`	0 – 42	Preemption mode
`GetTaskID()`	2	
`DisableAllInterrupts()`	4	
`EnableAllInterrupts()`	4	
`SuspendAllInterrupts()`	16	
`ResumeAllInterrupts()`	18	
`SuspendOSInterrupts()`	44	
`ResumeOSInterrupts()`	34	
`GetResource()`	50 – 76	Event support
`ReleaseResource()`	34	
`SetEvent()`	64 – 76	Arbitration rounds
`ClearEvent()`	18	
`GetEvent()`	4	
`WaitEvent()`	118 – 338	Arbitration rounds, schedule table support
`SetRelAlarm()`	22	
`CancelAlarm()`	4	
`StartScheduleTableRel()`	22	
`StopScheduleTable()`	14	

Operating system component	Size in B	Dependencies
IRQ table	64 – 576	Number of tasks
Initialization	78 – 1,348	Number of tasks, alarms, schedule tables
Resource prologue	102	
Task prologue	164 – 236	Task type, event support, timing protection, cell multiplexing
TT task wrapper	20 – 2,054	Event support, cell multiplexing
Explicit context switch	46	Needed for PIP resources

Table 6.2: Code size of the SLOTH operating system for different components and available system calls. The sizes are given for the SLOTH implementation on the Infineon TriCore platform; if a component depends on the application configuration, the size is given as a range together with a list of the dependent features.

operating system would otherwise have to provide in software. This reduction in the operating system software part affects both the code size and the data size.

6.2.1 Operating System Code Size

An overview of the SLOTH code footprint on the Infineon TriCore reference platform is shown in Table 6.2; since many parts of the operating system depend on configured features and the number of allocated application objects, the corresponding numbers are expressed in ranges where needed.

Most system call implementations are very concise; hence, they are subject to inlining into the application code to eliminate additional function calls with

corresponding register saving and restoring if the system is configured by the application developer to use inlining techniques. Further optimization possibilities result from the fact that in static applications, the system call parameters are also statically available to the compiler. The system call `ActivateTask()`, for instance, can in essence be compiled to a single write instruction to the corresponding memory-mapped interrupt request register; if–else cascades in system calls such as shown in Figure 5.3(b) are therefore automatically reduced by the compiler if it features the basic optimization technique of constant propagation. Additionally, inlined system calls can be further optimized by the compiler by eliminating common subexpressions, for instance. To state an upper boundary, the numbers in Table 6.2 reflect the worst-case sizes of exlined system calls without such optimizations.

Additional operating system components include the interrupt vector table, which is vital for task dispatching in SLOTH and whose size depends on the number of application tasks, and code for the initialization of the interrupt and timer hardware if necessary. In SLEEPY SLOTH and SLOTH ON TIME systems, additional task prologues and wrappers are needed for operation; in systems with PIP resources (see Section 4.3.7), an explicit context switch function is needed. Besides the actual operating system code, the start-up code as supplied by the compiler takes up 1,024 bytes in code memory, which can be stripped to 624 bytes of functionality actually needed by SLOTH systems.

The total code size numbers for tailored SLOTH operating systems heavily depend on the features configured by the application and the system calls it actually uses—in the presented evaluation, unused system calls are eliminated through whole-program optimization and function level linking support by the compiler and linker. The total code size numbers for the example applications introduced in Table 6.1 are shown in Table 6.3, together with the corresponding numbers for the commercial systems.

The SLOTH operating systems exhibit a lower code size than the commercial operating systems for all of the example applications. However, the ECC1, OSEKtime, and AUTOSAR SLOTH operating systems do not scale very well when the application configures lots of tasks, shown by the higher maximum numbers compared to the minimum numbers in Table 6.3. This is because SLOTH operating systems of those classes make use of task prologues, which are statically generated in a tailored variant for each of the application tasks although they have distinct but similar functionality.

The lower numbers in the range of SLOTH code sizes show the code sizes for the example applications using a modified SLOTH operating system source that is optimized for the total prologue code size by making use of prologue *functions* with run time parameters instead of generating prologues that are statically customized per task. In that variant, the SLOTH operating systems *do* scale well compared to the commercial operating systems in the BCC1, ECC1, and AUTOSAR classes. In the OSEKtime class, the commercial operating system shows almost no increase in code size in contrast to SLOTH ON TIME; this is because the

Application	Code size in B
Minimal BCC1 app	720 5,046
Maximum BCC1 app	1,024 7,038
Minimal ECC1 app	944–968 6,012
Maximum ECC1 app	2,208–4,824 7,506
Minimal OSEKtime app	944–960 2,160
Maximum OSEKtime app	1,864–1,888 2,210
Minimal AUTOSAR app	1,160–1,184 30,178
Maximum AUTOSAR app	4,192–7,224 34,082

Table 6.3: Code size of the SLOTH operating systems and the commercial operating systems for the example applications. The sizes for the example applications introduced in Table 6.1 are stated in bytes for the Infineon TriCore platform; the SLOTH operating system numbers are shown in bars, the commercial operating system numbers are shown in bars. The lower SLOTH code size numbers refer to an optimized operating system version that uses prologue functions. All sizes include the start-up code for the platform, which is 624 bytes in SLOTH systems.

commercial OSEKtime operating system only enlarges its dispatcher table held in ROM (see evaluation of operating system data size in Section 6.2.2) but SLOTH ON TIME extends its timer initialization code instead. In total, the modified SLOTH variant trades its reduced prologue code sizes for an increase in the task dispatch latencies due to dynamic decisions that have to be performed at run time depending on the affected task; these latencies are increased by between 5 and 15 clock cycles per dispatch (see also comprehensive evaluation of SLOTH latencies in Section 6.3).

In general, a minimal but feature-complete BCC1 SLOTH operating system takes 458 bytes of code memory for itself, which includes a minimal interrupt vector table, initialization code, and all system calls of the OSEK BCC1 conformance class (task, ISR, resource, and alarm system calls). Together with the compiler-supplied start-up code of 624 bytes, the total operating system image will consume 1,082 bytes in code memory.

6.2.2 Operating System Data Size

A critical part of an embedded system is its data usage in RAM, since having to add external RAM leads to significant cost increase. Internal RAM uses large parts of the chip transistor area and is therefore also very expensive; it has been reported that the cost is 8 to 16 times the cost of ROM [DMT00].

The data fields needed by the SLOTH operating system to execute also depend on the system configuration as specified by the application configuration as well as on the configuration of operating system objects such as tasks and resources; the data footprint therefore scales with the demand of the application. Table 6.4 lists the sizes for all data fields potentially needed by the operating system in the Infineon TriCore implementation, together with an annotation of the applicable configuration feature and a description of the field. If the size of a field depends on the number of configured objects, it is specified as a multiplication; this way, the exact amount of data memory needed can be calculated for any given application. On the TriCore, heavily accessed data fields are kept in global registers; read-only data can be kept in ROM or flash memory.

Table 6.5 shows the data memory footprint for the example applications introduced in Table 6.1, together with the corresponding numbers for the commercial operating systems. In addition, Table 6.6 shows the data fields used by SLOTH for the example applications. A minimal SLOTH application needs as few as 8 bytes of operating system memory in RAM, and the operating system demands scale with the number of system objects allocated by the application as well as with additional features configured by it. A minimal time-triggered application does not need more than 8 bytes of RAM either, and even a more full-featured time-triggered application does not increase this number a lot; this is because the schedule table and needed run time information is encapsulated in the timer hardware and its initialization instead of storing it in a table (compare to the increase in ROM size between the minimal and the maximum OSEKtime application in the commercial operating system).

All numbers do not include the memory demand for stacks and CSA stacks, since their allocation sizes depend on the actual application code that is executed on top of the operating system. In general, however, SLOTH systems will need less stack due to the potential for heavy system call inlining if configured by the application developer (see also Section 6.2.1); inlined system calls do not use the stack for saving and restoring registers and for passing parameters and return values. Systems with less preemption—that is, systems with non-preemptive task groups or completely non-preemptive systems—are also used with the purpose of saving stack memory resources, since less stack for preemption context has to be reserved. Additionally, basic task systems need less stack than extended task systems, which need one stack to be allocated per task for the whole system run time.

Size	Configuration	Field
4 B	Basic tasks	Current task ID (in global register)
4 B × #tasks	Basic tasks	Current task ID stack
4 B	Basic tasks	Current task ID pointer (in global register)
x B	Basic tasks	Common stack for all basic tasks
x B	Basic tasks	Common CSA stack for all basic tasks
1 B × #tasks	Multiple activ.	Activation counter per task
1 B × #resources	Resources	Resource stack
1 B	Resources	Resource stack pointer
1 B	ISRs	Category-1 ISR suspend counter
1 B	ISRs	Category-2 ISR suspend counter
1 B	ISRs	Category-2 ISR suspend priority
1 B × #tasks	Extended tasks	Mask of 8 set events per task
1 B × #tasks	Extended tasks	Mask of 8 waited-for events per task
4 B	Extended tasks	Mask of 32 `hasSeenCPU` flags
x B × #tasks	Extended tasks	1 stack per extended task
x B × #tasks	Extended tasks	1 CSA stack per extended task
4 B × #tasks	Extended tasks	1 CSA stack pointer per extended task
4 B × #tasks	Extended tasks	Array of SRNs per task for re-triggering at run time (in ROM)
4 B	Extended tasks & basic tasks	1 common CSA stack pointer for all basic tasks
5 B	Extended tasks & basic tasks	Stack and ID information when extended tasks interleave basic task execution
1 B × #tasks	Extended tasks & resources	Dispatching information about which task currently possesses which resource
1 B × #PIP res.	PIP resources	Locking task per PIP resource
1 B × #PIP res.	PIP resources	Waiting task per PIP resource
1 B	Schedule tables	Current schedule table ID
4 B × #tasks	Schedule tables & execution budgets	Execution budget per task (in ROM)
8 B × #tasks	Schedule tables & execution budgets	Array of timer registers per task for re-configuration at run time (in ROM)
1 B × #tasks	Schedule tables & task multiplexing	Index to current task activation per task
2 B × #task activ.	Schedule tables & task multiplexing	Counter values for all task activations per task (in ROM)
1 B × #tasks	Schedule tables & deadl. multiplexing	Index to current deadline per task
2 B × #deadlines	Schedule tables & deadl. multiplexing	Offset values for all deadlines per task (in ROM)

Table 6.4: Data fields allocated by SLOTH and their sizes. The data field sizes are shown for the Infineon TriCore platform and depend on the system configuration and the number of configured application objects (e.g., tasks). Some of the data fields are held in global registers (if heavily accessed) or in ROM (if read-only).

6.3 Performance Microbenchmarks

In order to assess the effects of the hardware-centric SLOTH design on the performance and latencies induced by the operating system, comprehensive timing measurements were performed on the SLOTH reference platform, the Infineon TriCore TC1796b. The chip was clocked at 50 MHz both for its system frequency and its CPU frequency (corresponding to a clock cycle time of 20 ns), and the mea-

Application	Data size in RAM in B	Data size in ROM in B
Minimal BCC1 app	8 56	0 56
Maximum BCC1 app	42 104	0 136
Minimal ECC1 app	25 56	8 60
Maximum ECC1 app	140 176	68 224
Minimal OSEKtime app	8 32	0 40
Maximum OSEKtime app	37 40	0 144
Minimal AUTOSAR app	38 240	16 2,268
Maximum AUTOSAR app	142 1,496	288 4,008

Table 6.5: Data usage of the SLOTH operating systems and the commercial operating systems for the example applications. The table shows the data usage in bytes for the example applications introduced in Table 6.1 on the Infineon TriCore platform, differentiated between writable data (RAM) and read-only data (ROM); the SLOTH numbers are shown in bars, the commercial operating system numbers are shown in bars. The sizes do not include stack sizes and CSA stack sizes, both of which depend on the actual application code.

surements were performed using a Lauterbach PowerTrace hardware debugging and tracing unit and averaged over at least 1,000 iterations each. The numbers are stated in frequency-independent clock cycles. All programs were executed from internal no-wait-state RAM to avoid caching effects. In some situations, the distribution of the samples exhibited two distinct peaks of similar height, which are located exactly 4 cycles apart and which is presumably related to unstableness in measuring conditional jumps. Aside from this effect, the deviations from the average have shown to be negligible in the measurements.

The performance measurement results are important for the application developer when designing his real-time system, since he has to take system overhead into account. For preemptive static-priority systems such as implemented by SLOTH, the effective worst-case execution time (WCET) of each task can simply be calculated by taking the application task WCET and adding two times the context switch overhead—for dispatching and terminating the task [Liu00]. With this overhead-aware WCET, the regular schedulability analysis can be performed, reaching higher schedulability with lower system overhead. Other than that, SLOTH does not incur any overhead that the application cannot plan for.

All performance benchmarks were executed on SLOTH and on the commer-

Application	SLOTH usage
Minimal BCC1 app	Task ID stack
Maximum BCC1 app	Task ID stack, ISR suspend counters, resource stack
Minimal ECC1 app	Event masks and interleaving info, stack pointers
Maximum ECC1 app	Event masks and interleaving info, stack pointers, resource stack and info, ISR suspend counters
Minimal OSEKtime app	Task ID stack
Maximum OSEKtime app	Task ID stack, current schedule table ID
Minimal AUTOSAR app	Event masks and interleaving info, stack pointers, current schedule table ID
Maximum AUTOSAR app	Event masks and interleaving info, stack pointers, resource stack and info, ISR suspend counters, current schedule table ID

Table 6.6: Data fields allocated by the SLOTH operating system for the example applications. The table shows the data fields needed by the SLOTH implementation on the Infineon TriCore for the example applications introduced in Table 6.1.

cial OSEK, OSEKtime, and AUTOSAR operating systems, depending on the nature of the benchmark. By using the very same benchmarking applications employing the standard interfaces, the results can be directly compared. Note that the commercial operating systems use a traditional software scheduler for event-triggered and time-triggered task scheduling and dispatching. Since SLOTH tasks are implemented using the interrupt subsystem, for instance, SLOTH task system calls bear latencies incurred by the hardware in addition to latencies incurred by the SLOTH software. The SLOTH numbers discussed in this section include all hardware-related preemption costs such as waiting for the bus arbitration (see also Section 5.5.1) and therefore reflect the worst-case latencies as perceived by the real-time application.

6.3.1 OSEK OS BCC1 SLOTH Systems

The left part of Table 6.7 and Table 6.8 shows the number of CPU cycles it takes to execute OSEK OS system calls in an OSEK BCC1 system (i.e., in a system with only basic tasks) on the SLOTH operating system (bars) and the commercial OSEK operating system (bars). Depending on the current system state, a system call can lead to a task dispatch or not—for instance, activating a task with a lower priority does *not* lead to a dispatch, whereas activating a task with a higher priority *does* lead to a preemption in a preemptive system. In those cases, both possibilities are listed with their respective latencies. Additionally, a *range* of clock cycles is specified for the SLOTH numbers when the corresponding system call depends on the interrupt arbitration latency of the hardware platform; in those cases, SLOTH has to use `nop` timing to wait for the arbitration (see Sec-

System call	Disp.	BCC1 cycles	ECC1 cycles
`ActivateTask()`	no	34–40 76	34–40 88
`ActivateTask()`	yes	56–62 284	115–121 292
`TerminateTask()`	yes	13 284	81-87 269
`ChainTask()`	yes	61–67 353	110-116 399
`Schedule()` (preemptive)	no	0 7	0 7
`Schedule()`	no	12 32	12 33
`Schedule()`	yes	29 234	89 233
`GetTaskID()`	no	2 14	2 14
`DisableAllInterrupts()`	no	2 11	2 11
`EnableAllInterrupts()`	no	2 23	2 23
`SuspendAllInterrupts()`	no	4 19	4 18
`ResumeAllInterrupts()`	no	8 29	8 30
`SuspendOSInterrupts()`	no	14 18	14 18
`ResumeOSInterrupts()`	no	12 29	12 29

Table 6.7: OSEK OS task and interrupt system call latencies. The numbers show the OSEK OS system call latencies on the Infineon TriCore platform in basic task systems (BCC1; do *not* need stack switches) and extended task systems (ECC1; *do* need stack switches when dispatching another task), with settings that do or do not involve task dispatches (disp.). The SLOTH operating system numbers are shown in bars, the commercial operating system numbers are shown in bars.

tion 5.5.1), and the given numbers specify the best case (1 arbitration round) and the worst case (4 arbitration rounds), respectively.

Due to their very low software footprint (see also Section 6.2.1), the SLOTH system calls are also very fast in their execution. A task switch can be as fast as 56 clock cycles when synchronously activating and dispatching a high-priority task or even 13 clock cycles for terminating a task and returning to the previously executing one; the numbers reflect the execution time from the start of the system call to the first user instruction in the application task. Activating a low-priority task without a direct dispatch is also affected by interrupt arbitration synchronization of the operating system (see Section 5.5.1), so the latency depends on the system configuration (see specified latency ranges in Table 6.7 and Table 6.8). Most of the other system calls alter the CPU priority in their implementations (e.g., `GetResource()` raises it to the resource ceiling priority), which

System call	Disp.	BCC1 cycles	ECC1 cycles
`SetEvent()` not waited for	no	*n/a*	7 44
`SetEvent()`	no	*n/a*	43–49 83
`SetEvent()`	yes	*n/a*	118–125 215
`ClearEvent()`	no	*n/a*	5 36
`GetEvent()`	no	*n/a*	2 14
`WaitEvent()` already set	no	*n/a*	10 39
`WaitEvent()`	yes	*n/a*	118–124 212
`GetResource()`	no	14 71	21 71
`ReleaseResource()`	no	10 127	10 122
`ReleaseResource()`	yes	32 283	91 287
`SetRelAlarm()`	no	16 44	16 44
`CancelAlarm()`	no	1 37	1 37

Table 6.8: OSEK OS event, resource, and alarm system call latencies. The numbers show the OSEK OS system call latencies on the Infineon TriCore platform in basic task systems (BCC1; do *not* need stack switches) and extended task systems (ECC1; *do* need stack switches when dispatching another task), with settings that do or do not involve task dispatches (disp.). The SLOTH and SLEEPY SLOTH operating system numbers are shown in bars, the commercial operating system numbers are shown in bars.

is independent of the number of arbitration cycles; the corresponding system call run times are therefore static.

Compared to the commercial operating system, SLOTH performs better in all cases. Especially the benchmarks that include a scheduling and dispatch operation are significantly faster on SLOTH, which relies on the interrupt system to perform these tasks; the speed-up numbers compared to the commercial operating system go up to a factor of 37. Those system calls that are not implemented in a more hardware-centric way in SLOTH compared to traditional operating systems (e.g., the interrupt system calls) have a performance similar to the software-based operating system.

Depending on the hardware configuration and on whether a new task is activated or a running one terminates, SLOTH only needs between 260 ns and 1,320 ns for a task switch (including the actual context switch) on a 50 MHz system. The idle CPU overhead in SLOTH is zero, since there is no base load for maintaining alarm ticks. This is due to the static pre-configuration of the hardware timer system for each employed alarm (see Section 4.2.4).

In an example application, the operating system boot time between executing the reset vector and entering the first auto-started user task is 2,651 clock cycles for SLOTH and 1,798 cycles for the commercial OSEK OS operating system. These numbers do not include the initialization of the CSA context stack, whose length depends on the number of CSAs needed by the application (see also Section 5.5.2). Since SLOTH moves information from the productive run time phase to the initialization phase, which is only executed once (see also Section 3.1), this phase takes longer in order to program the interrupt subsystem and the timer subsystem if necessary.

When comparing the interrupt dispatch latencies, the commercial operating system and SLOTH both only execute the necessary instructions to directly invoke the associated ISR. Dispatching either a category-1 ISR or a category-2 ISR takes the commercial operating system 14 clock cycles, whereas it takes SLOTH 12 clock cycles for both cases. For the worst-case interrupt latency, however, the real-time engineer also has to consider the worst-case code section that the operating system synchronizes by disabling interrupts, since these sections can delay ISR execution. There are very few of those sections in the SLOTH operating system (see also Section 5.5.1), and all of them are very short and bounded in the times that they lock the operating system. Considering the complete functionality provided by SLEEPY SLOTH systems with extended tasks (see details in Section 6.3.2), the worst-case execution time of the longest critical section in SLEEPY SLOTH amounts to 57 to 63 clock cycles, depending on the configuration of the interrupt arbitration system (see Section 5.5.1); this critical section occurs when an extended task blocks while waiting for an event (see Figure 5.6(b)). All of the other few critical sections in the operating system amount to shorter periods of interrupt locking.

As argued in the problem analysis in Section 2.2, traditional systems possess restrictions related to uncombinable properties of control flows, which is exemplified here with an operating-system-scheduled task that needs to be activated asynchronously by a hardware event. In a traditional system, this needs to be worked around by attaching a category-2 ISR to the hardware event and by having the ISR activate the corresponding task. In the commercial OSEK OS system, this scenario takes 1,142 clock cycles from the hardware IRQ to the execution of the task—including the `ActivateTask()` system call in the ISR, the ISR termination, and the task dispatch. In SLOTH, the universal control flow abstraction allows for asynchronously activated tasks, which are directly dispatched by the hardware within only 12 clock cycles, resulting in a speed-up of 95 x. Additionally, these ISR-like control flows can be flexibly used with blocking semantics if desired, and they can participate in the priority ceiling protocol together with synchronously activated control flows.

6.3.2 OSEK OS ECC1 SLEEPY SLOTH Systems

Applications that make use of extended tasks have to be provided with the ability to block while waiting for an event. Additionally, extended tasks run on stacks

and CSA stacks of their own; thus, corresponding stack switches have to be performed by the task prologues in SLEEPY SLOTH (see Section 4.3.1), which has an impact on the latency of the affected system calls. The performance of the system calls in ECC1 systems is shown in the right part of Table 6.7 and Table 6.8; the SLEEPY SLOTH numbers are depicted in  bars, the numbers for the commercial OSEK operating system are depicted in  bars. The additional software checks and stack switches in the task prologues in SLEEPY SLOTH induce additional overhead in those microbenchmarks that involve dispatching a new task—`ActivateTask()`, `TerminateTask()`, `ChainTask()`, `Schedule()`, and `ReleaseResource()`. Comparable latencies are achieved by the additional ECC1 system calls if they involve dispatching a new task—`SetEvent()` for a higher-priority blocked task as well as `WaitEvent()`.

In some scenarios, however, the way that SLEEPY SLOTH handles task terminations (see Section 4.3.2) can actually lead to *fewer* context switches compared to SLOTH. Consider a low-priority task `Task1` being preempted by a high-priority task `Task3`, which then activates a mid-priority task `Task2`. When `Task3` terminates, SLOTH issues a return-from-interrupt instruction, which first restores `Task1`'s context before being immediately preempted by `Task2`, leading to an additional context save. In SLEEPY SLOTH, `Task3` terminates by lowering the CPU priority to zero, which has `Task2` preempt it immediately because of its higher priority, without an additional context save and restore.

Although SLEEPY SLOTH also uses the hardware interrupt controller, whose run-to-completion model does not fit well the semantics of blocking threads as implemented by SLEEPY SLOTH, it can still keep the total system call latencies low. However, SLEEPY SLOTH switches between extended tasks are considerably slower (81–125 cycles) than SLOTH task switches between basic tasks (13–67 cycles) due to the stack switches and involved decisions. The comparison shows the benefit of SLOTH being tailorable to the application; if the application does not need the flexibility of blocking extended tasks, the operating system latencies can be kept significantly lower. Despite the additional overhead compared to BCC1 SLOTH system calls, SLEEPY SLOTH system calls are still considerably faster than those in the commercial operating system with a software scheduler; the speed-up numbers go as high as 3.6 x for system calls that involve a dispatch operation. Note that the software-based commercial implementation shows about the same overhead for switches between basic tasks as for switches between extended tasks and therefore does not tailor its operating system functionality to the needs of the application.

In mixed systems with both kinds of tasks, SLEEPY SLOTH's design goal was to not incur considerable overhead for task switches between basic tasks only (see Section 4.3.5). On application level granularity, this is shown by comparing the task switch times in systems with only basic tasks to those in systems with only extended tasks (see Table 6.7 and Table 6.8); since the SLOTH operating system is completely tailorable to the application configuration, it avoids unnecessary overhead in basic-task-only systems. For an analysis on task level granularity,

	... to basic task	... to extended task
Switch from basic task...		
`ActivateTask()`	75–81 287	111–117 294
`TerminateTask()`	27 321	91 310
`ChainTask()`	92–98 360	111–117 369
`SetEvent()`	*n/a*	114–120 215
`WaitEvent()`	*n/a*	*n/a*
`ReleaseResource()`	53 286	87 325
Switch from extended task...		
`ActivateTask()`	125–131 287	113–121 292
`TerminateTask()`	83–89 315	79–87 269
`ChainTask()`	101–106 361	108–116 399
`SetEvent()`	*n/a*	116–125 215
`WaitEvent()`	119–125 221	116–124 212
`ReleaseResource()`	102 295	91 287

Table 6.9: OSEK OS system call latencies for mixed-type task switches. The numbers show the OSEK OS system call latencies in clock cycles on the Infineon TriCore platform in systems that involve task switches in four configurations of switches *from* a basic or an extended task *to* a basic or an extended task. The SLEEPY SLOTH numbers are shown in bars, the commercial operating system numbers are shown in bars.

Table 6.9 shows the results of the evaluation of a *mixed* SLEEPY SLOTH system running both kinds of tasks in one application, generating different kinds of task switches. All of the system calls that involve task switches are shown in four variations where applicable—switches *from* a basic or an extended task *to* a basic or an extended task.

System calls that generate task switches between two basic tasks (upper left part of Table 6.9) are on SLEEPY SLOTH's low end with clock cycle numbers between 27 and 98; those values compare to clock cycle numbers between 13 and 67 in basic-only systems (see Table 6.7 and Table 6.8). Basic task scheduling in SLEEPY SLOTH is therefore burdened by an overhead of 14 to 31 cycles, added to a base overhead that is already very low. Thus, in a mixed task system, the task switch overhead scales with the demand of the involved tasks—with cheaper task switches between basic tasks than between extended tasks, which need additional stack switches. Compared to the commercial OSEK operating system,

```
<handlerTTTask2>:
    mov %d0,2944
    mtcr $psw,%d0              // enable global address registers
    isync                      // synchronize previous instruction
    st.a [+%a9]-4,%a8          // save preempted task ID on stack
    mov.a %a8,2                // set new task ID
    st.t <GPTA0_LTCCTR11>,3,1  // enable deadline cell
    bisr 2                     // set exec prio 2, save context, enable IRQs

    call userTTTask2           // enter user code

    disable                    // suspend IRQs for synchronization
    st.t <GPTA0_LTCCTR11>,3,0  // disable deadline cell
    rslcx                      // restore context
    ld.a %a8,[%a9+]            // restore preempted task ID from stack
    rfe                        // return from interrupt handler (re-enables IRQs)
```

Figure 6.1: Compiled time-triggered task interrupt handler in SLOTH ON TIME. The shown machine code is generated for the Infineon TriCore platform for an example task `TTTask2` with one deadline (see example application configuration in Table 4.5).

SLEEPY SLOTH system calls are also faster in the mixed task scenario, with speed-up numbers up to a factor of 12.

6.3.3 OSEKtime-Like SLOTH ON TIME Systems

In OSEKtime-like time-triggered systems, the task activation and dispatch points are statically configured before compile time and used by SLOTH ON TIME to configure the hardware timer cells appropriately or used by traditional operating systems to provide a dispatch table in read-only memory. Table 6.10 shows the latencies induced by the SLOTH ON TIME operating system (bars) and the commercial OSEKtime operating system (bars) when dispatching a time-triggered task and when terminating it. The base overhead in SLOTH ON TIME is 12 cycles for both dispatching and terminating a task.

The dispatch number of 12 cycles includes *all* costs between the preemption of the running task or the idle loop to the first user instruction in the dispatched task (see assembly instructions of an example wrapper in Figure 6.1). Thus, this number reflects the complete SLOTH ON TIME prologue wrapper as introduced in Section 4.4.3 and as shown in Figure 5.5. Since an OSEKtime-like system is strictly stack-based, all tasks run on the same stack, so the wrapper only has to save the context of the preempted task on the stack. The SLOTH overhead numbers of 56 to 62 cycles when activating a task with preemption (see Table 6.7) exactly correspond to the 12 cycles for the context save prologue plus the overhead for the `ActivateTask()` system call executed before the preemption. Since with time-triggered activations in SLOTH ON TIME that system call overhead does not apply, the dispatch operation yields the very low total number of 12 cycles.

When using deadline monitoring, the base dispatch and termination overhead of 12 cycles increases by about 12 cycles per registered deadline of that task

	Activation muxing disabled	Activation muxing enabled
Time-triggered task dispatch		
0 deadlines		
... from OSEKtime task	12 / 108	25 / 108
... from basic task	12 / 108	26 / 108
... from extended task	28 / 108	40 / 108
1 deadline	24 / 108	37 / 108
2 deadlines	36 / 108	44 / 108
3 deadlines	48 / 108	53 / 108
4 deadlines	56 / 108	61 / 108
n muxed deadlines	12 / 108	25 / 108
Time-triggered task termination		
0 deadlines		
... to OSEKtime task	12 / 36	12 / 36
... to basic task	12 / 36	12 / 36
... to extended task	12 / 36	12 / 36
1 deadline	24 / 36	24 / 36
2 deadlines	36 / 36	36 / 36
3 deadlines	44 / 36	44 / 36
4 deadlines	56 / 36	56 / 36
n muxed deadlines	25 / 36	25 / 36

Table 6.10: Latencies of time-triggered task dispatch and termination in OSEKtime OS systems. The numbers show the latencies in clock cycles on the Infineon TriCore platform in systems with varying numbers of deadlines per task, including SLOTH ON TIME system variants with deadline multiplexing and activation cell multiplexing (right column). The SLOTH ON TIME numbers are shown in bars, the commercial operating system numbers are shown in bars.

(see Table 6.10). This stems from the fact that SLOTH ON TIME activates and deactivates the deadline cells for that task (see also Section 4.4.4), which compiles to a single memory-mapped store instruction (see Figure 6.1). Due to memory bus synchronization, this instruction takes about 12 clock cycles. If deadline multiplexing as described in Section 4.4.8 is used instead, no additional overhead is

incurred during dispatch, but an increase of 13 cycles can be measured for task termination, representing the cost of maintaining the state of the offset array and re-configuring the deadline cell. However, this increased overhead does not increase further with additional deadlines to be monitored for the same task; starting with two deadlines per task, multiplexing yields a performance advantage. This advantage is traded for a slight increase in memory footprint for the offset array and the associated index variable (see also Table 6.4).

If timer cells are sparse, activation cell multiplexing can be used (see Section 4.4.8); the corresponding dispatch latencies are shown in the right column of Table 6.10. Multiplexing an activation cell totals to about 5 to 14 cycles per dispatch, depending on the number of deadlines of a task; no additional overhead is incurred at task termination. The multiplexing overhead is only applicable to tasks with multiple activations in a single dispatcher round; tasks that are activated only once per dispatcher round do not need to be multiplexed and therefore retain the usual overhead.

If SLOTH ON TIME is run in a mixed-mode system together with event-triggered tasks running in the background, the dispatch overhead depends on the type of task that is preempted by the time-triggered OSEKtime task. Preempting an event-triggered basic task does not add to the dispatch overhead, whereas preempting an extended task adds 16 cycles for the required stack switch, totaling to a low latency of 28 cycles for the worst-case scenario in a mixed-mode system. Task termination is not affected by the underlying event-triggered system since the restore operation of the stack pointer is encapsulated in the return-from-interrupt instruction on the TriCore platform.

The commercial OSEKtime operating system does not feature implementation variants with deadline or activation multiplexing, since it uses a traditional dispatch table held in data memory. Thus, its dispatch and termination overhead is always 108 and 36 cycles, respectively—even in its general-purpose variant with both activation and deadline multiplexing, SLOTH ON TIME therefore has a speed-up of 4.3 and 1.4, respectively. Additionally, the commercial OSEKtime operating system introduces one additional IRQ per deadline per dispatcher round, executing for 95 cycles to check whether the corresponding deadline has been violated, adding to the total overhead as perceived by the application. Due to its way of managing deadlines, SLOTH ON TIME can prevent those IRQs and the corresponding overhead, which is further analyzed and discussed in Section 6.5.2. In total, when comparing SLOTH ON TIME to the commercial operating system, the number of additionally available clock cycles per dispatcher round is 120 per task activation point without a deadline, plus 107 per task activation with one or more associated deadlines, plus 95 per monitored deadline.

6.3.4 AUTOSAR-OS-Like SLOTH ON TIME Systems

AUTOSAR OS systems are event-triggered with static task priorities, but tasks can be activated in a time-triggered manner based on statically configured schedule

	Budgeting disabled	Budgeting enabled
Time-triggered task act.		
... of basic task	14 1,376	33 1,679
... of extended task	68 1,376	87 1,679
`TerminateTask()`		
... from basic task	13 346	39 529
... from extended task	76 346	112 529
`StartScheduleTableRel()`	24–91 630	24–91 630
`StopScheduleTable()`	9 275	9 275

Table 6.11: Time-based AUTOSAR OS task activation and termination latencies and corresponding system call latencies. The numbers show the latencies in clock cycles on the Infineon TriCore platform; the SLOTH ON TIME numbers are shown in bars, the commercial operating system numbers are shown in bars.

tables. The latencies required in SLOTH ON TIME to activate a task in such a manner and to terminate it using the system call `TerminateTask()` are shown in Table 6.11. Depending on the type of task—whether it is a basic run-to-completion task or an extended blocking task—the activation latency is 14 or 68 cycles, and the termination latency is 13 or 76 cycles, respectively. The variance stems from the necessity to switch stacks or not. The commercial operating system uses 1,376 clock cycles to activate a task in a time-triggered manner, plus 346 cycles to terminate it, resulting in a SLOTH ON TIME speed-up of 106 x and 5 x, respectively.

Execution time budgeting is employed in SLOTH ON TIME systems by starting, stopping, and resetting a dedicated budget timer cell upon task switches (see Section 4.4.7). The total dispatch overhead with execution budgeting is shown in the right column of Table 6.11. The additional overhead incurred by the budgeting mechanism therefore totals to 19 to 36 clock cycles, depending on the type of dispatch (activation or termination) and involved task. To implement budgeting, the commercial AUTOSAR operating system needs an additional 303 and 183 clock cycles to activate and terminate a task, respectively, resulting in a minimum budgeting speed-up for SLOTH ON TIME of 16 x and 5 x, respectively.

The system call to stop a schedule table at run time takes 9 cycles (see lower part of Table 6.11); starting a schedule table with a relative offset at run time takes between 24 and 91 clock cycles for tables with up to two task activations. The actual number also depends on the fact whether a global control cell is used for that schedule table or not (see also Section 4.4.3); enabling a global control cell has a constant overhead, whereas enabling all involved activation cells scales

linearly. In any case, re-configuring the timers using the offset specified at run time scales linearly with the number of involved timer cells; the rest of the timer cell state is pre-configured during the initialization. The commercial operating system needs 630 cycles to start and 275 cycles to stop a comparable schedule table, resulting in speed-up numbers of 7 x and 31 x, respectively.

When comparing SLOTH ON TIME to the commercial AUTOSAR OS system, the total benefit in clock cycles is at least 1,578 per task activation, plus 539 per schedule table start system call, plus 266 per schedule table stop system call, plus the SLOTH and SLEEPY SLOTH benefit for the regular event-triggered operation (see Section 6.3.1 and Section 6.3.2).

6.4 Lines of Operating System Code

The SLOTH design, which uses available hardware instructions to direct the hardware system to accomplish operating system tasks, also yields a concise operating system in terms of its source code. This is especially important in safety-critical systems that need its application and system code to be certified for correctness, since increased source code size and complexity leads to a higher probability for the occurrence of program errors due to increased error density [McC04] (see also Section 7.1.1). The size of the SLOTH source code that would need to be certified for an embedded system on the Infineon TriCore TC1796 is shown in Figure 6.2 (compare to Figure 5.2, which shows the abstract operating system parts in the build system); the numbers were obtained using the Unified Code-Count tool UCC 2011.10 [NDRTB07; NSP12] for both the C code and the Perl code. The considered size depends on several factors:

1. The class of the operating system and the system services it provides. SLOTH can provide a basic OSEK BCC1 operating system (see Section 4.2), an extended OSEK ECC1 variant (SLEEPY SLOTH; see Section 4.3), a time-triggered variant (SLOTH ON TIME; see Section 4.4), or a mixed combination of all of those systems plus additional AUTOSAR features like execution budgeting (see Section 4.4.7). The SLOTH source code base can be analyzed per class; Figure 6.2 shows the lines of code (LOC) for each of the mentioned classes.

2. Whether the whole product line or only one configured operating system is considered. The whole product line consists of Perl template files (upper third of Figure 6.2), which in turn generate the application-specific C files that make up the configured operating system (middle third). This includes the flexible HAL instance, which is tailored to the application to produce the residual HAL (see Figure 3.3).

3. The number of system objects that the application allocates. If the configured operating system variant is considered, its source code depends

on the application configuration, since it needs to allocate system objects like task identifiers and interrupt handlers. Thus, the LOC numbers in the application-specific part of Figure 6.2 represent numbers for a minimal application and a comprehensive application for each class as introduced in Table 6.1.

In any case, the static, application-independent code needs to be considered for the complete SLOTH code base (lower third of Figure 6.2). A configured SLOTH BCC1 operating system can total at less than 500 lines of code, which makes it a feasible target for certification. If the resulting operating system is inlined manually, a minimal implementation can even be put into less than 200 lines (see complete implementation in Figure 6.7 at the end of this chapter). If the operating system product line itself were to be certified, in addition to the template code this would entail certifying common calculations and the generator itself (upper part of the middle column in Figure 6.2). The verification of the application configuration can include an arbitrary number of sanity checks to ensure a consistent configuration to be fed into the SLOTH generator.

The source code *complexity* of the involved files is generally rather low. Most system services merely include sequential code with an occasional conditional branch (if–else block); except for the idle loop and the task loops that reset the stack pointer before executing the next instance of a task, no loops at all are included in the system code. The architecture-specific parts, however, include instructions that alter the control flow on the CPU, of course—like a return-from-interrupt instruction when terminating a task, for instance.

The Perl templates that generate the application-specific C code *do* include loops that iterate over the configured number of system objects to produce corresponding parts of the operating system (see also partial evaluation in the SLOTH flexible HAL in Figure 3.3). The used code patterns, however, repeat themselves throughout the templates; they mainly entail template code that generates C code for a system object depending on its properties (e.g., a different IRQ entry is generated for a basic task than for an extended task) and depending on the *system* configuration (e.g., timer initialization code is only generated if support for alarms, schedule tables, or execution budgeting is enabled).

Independent of the consideration of the SLOTH source code, the corresponding C compiler and the Perl interpreter (only if the whole SLOTH program family is considered) would also need to be certified. As elaborated, the SLOTH system itself is a welcoming subject for certification due to its conciseness and simplicity.

6.5 Rate-Monotonic Priority Inversion

Traditional operating systems exhibit a special type of priority inversion when they interrupt high-priority tasks to execute low-priority ISRs; this phenomenon is called rate-monotonic priority inversion. This problem is inherent to such system designs since they execute tasks at interrupt level zero, being interruptible

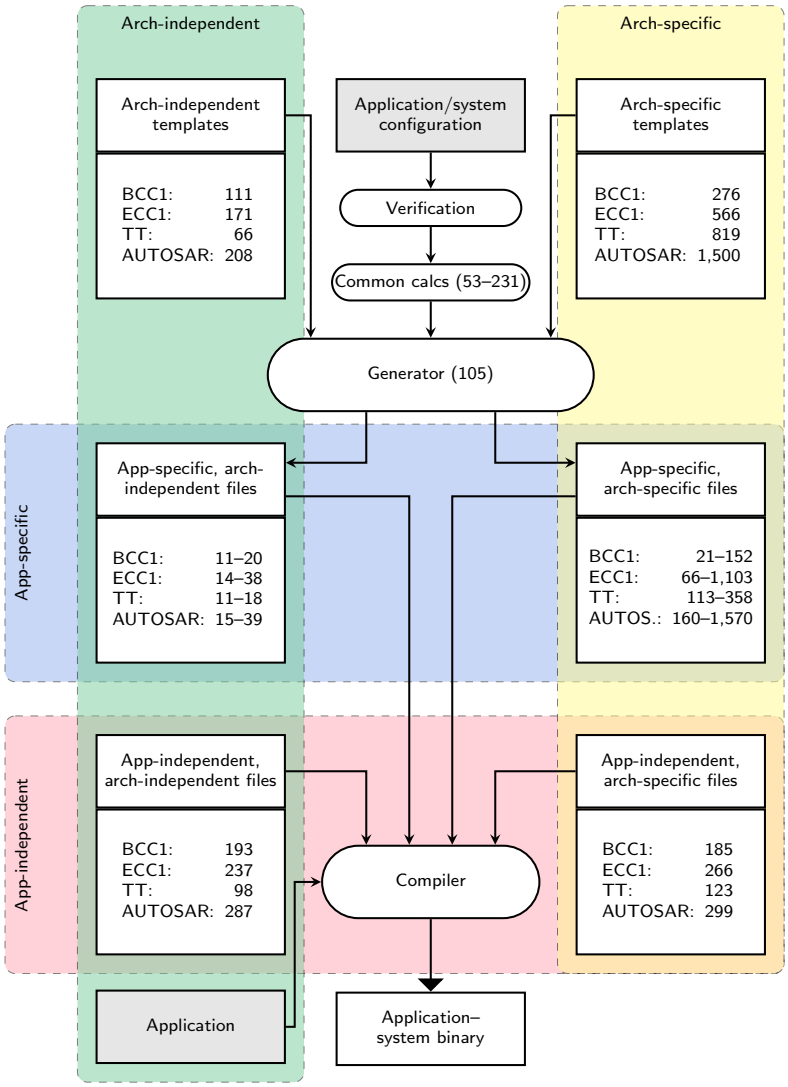


Figure 6.2: Lines of code in the different source parts of the SLOTH operating system. The figure shows the lines of code for an OSEK BCC1, an OSEK ECC1, a time-triggered (TT), and an AUTOSAR-like system with mixed features. The lines of code in the application-specific sources depend on the configuration; the given values correspond to a minimal and a comprehensive application. For an overview of the included operating system parts, see Figure 5.2.

by any hardware interrupt. The SLOTH operating systems run tasks as ISRs internally, executing them at the interrupt level corresponding to the task priority to keep low-priority ISRs from interrupting them. This section shows different situations of unbounded priority inversion and how SLOTH overcomes them by design.

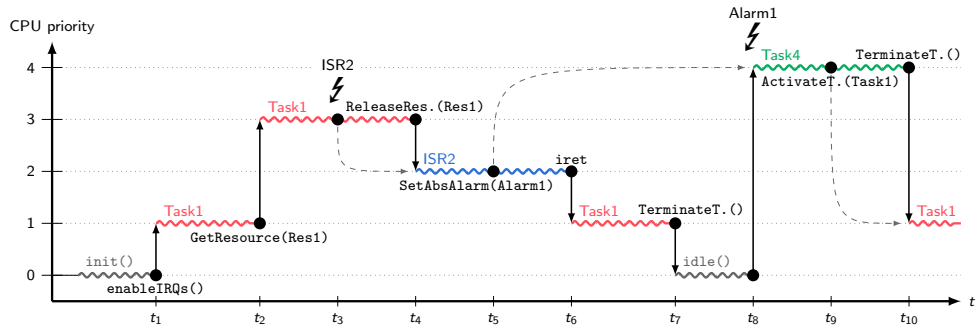
6.5.1 Priority Inversion in Event-Triggered Systems

The original situation of rate-monotonic priority inversion as introduced by Leyva-del-Foyo et al. [FMAN06a] occurs in event-triggered systems with static priorities whenever a task with a high semantic priority is interrupted and disturbed by an ISR with a low semantic priority. This situation is exemplified in the control flow depicted in Figure 6.3(b) when at t_3, `ISR2` with a semantic priority of 2 interrupts the execution of `Task1`, which has a semantic priority of 3 at that point due to a critical section guarded with a stack-based priority ceiling protocol. The interruption makes `Task1` leave the critical section delayed at t_6, potentially delaying other high-priority tasks.

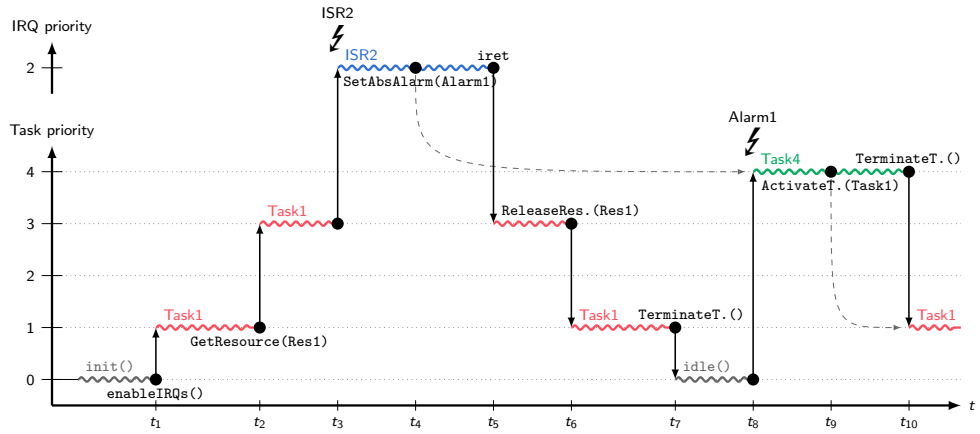
The same situation happening in a SLOTH system is depicted in Figure 6.3(a), where the dispatching of `ISR2` is deferred until after `Task1` leaves its critical section at t_4. This behavior corresponds to the configured *semantic* priorities of the application and increases the potential to analyze the application in advance, since priorities are never inverted in an unbounded manner in SLOTH systems.

This conceptual difference between SLOTH and traditional operating systems and the implications on situations of priority inversion can also be observed in running systems. Figure 6.4 shows execution traces of an application with a low-priority task `Task1`, preempted by a high-priority task `Task3`; during the execution of `Task3`, a medium-priority interrupt `ISR2` occurs. In the SLOTH system, the execution of `ISR2` is deferred until after `Task3` terminates (see Figure 6.4(b)). In the same application running on the commercial OSEK OS operating system, `Task3` is interrupted by `ISR2` (see Figure 6.4(a)), potentially making `Task3` miss its deadline due to this situation of rate-monotonic priority inversion.

Other solutions to that problem suggested by previous research include adapting the IRQ masks or the IRQ level when dispatching a task, depending on the configured semantic control flow priorities [FMAN06a; FMAN12; DGG10] (see comprehensive discussion of related work in Section 7.3.1). Such approaches prevent low-priority ISRs to interrupt high-priority tasks, but they do not dispatch high-priority tasks that are activated during the execution of a low priority ISR; this would need a further modification of the scheduler. In contrast to SLOTH, these approaches trade the elimination of rate-monotonic priority inversion for an increased operating system overhead. This overhead is incurred by the need to set the IRQ masks or the IRQ level in the system software, by having to maintain a consistent task priority in parallel, by incurring unnecessary context switches due to separate scheduling of tasks in software and of IRQs in hardware, and by the scheduler enhancement mentioned above. SLOTH, on the other hand, does not feature that trade-off; in contrast, it features superb performance and latency traits as evaluated in Section 6.3.



(a) Example control flow trace of the application introduced in Table 4.1 in a SLOTH system with a single priority space (see also Figure 4.1).

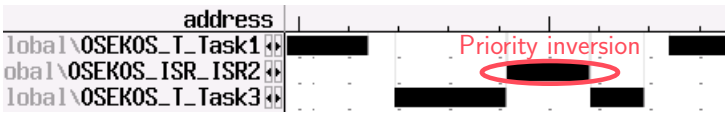


(b) Example control flow trace of the application introduced in Table 4.1 in a traditional system with two priority spaces.

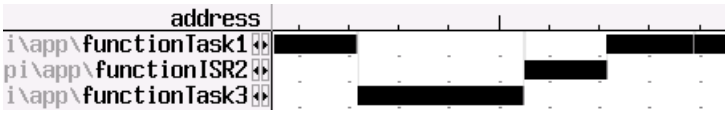
Figure 6.3: Example control flow traces of an application susceptible to rate-monotonic priority inversion. The traces show execution examples for the application introduced in Table 4.1. In traditional systems, the two priority spaces for tasks and IRQs lead to rate-monotonic priority inversion when the medium-priority `ISR2` interrupts the execution of `Task1`, which has a higher semantic priority at that time (a). In the SLOTH system, the single priority space leads to the execution of `ISR2` after `Task1` leaves its critical section, which reduces `Task1`'s priority to a low level (b).

6.5.2 Deadline Check Interrupts

OSEKtime-like systems with active deadline monitoring are usually implemented using deadline check interrupts. Those interrupts are triggered by a timer at the specified deadlines and check whether the corresponding task is still running, and—if so—issue a deadline violation to the application. For non-violated deadlines, which are the common case in a running system, this kind of implementation bears two problems. For one, if the deadline check interrupt is triggered



(a) Execution trace of a medium-priority interrupt occurring during execution of a high-priority task in the commercial OSEK OS system (see also Figure 1.2).



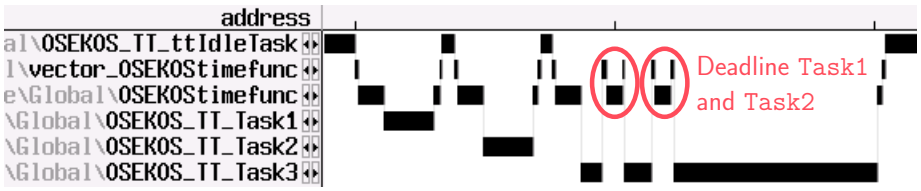
(b) Execution trace of a medium-priority interrupt occurring during execution of a high-priority task in SLOTH.

Figure 6.4: Prevention of rate-monotonic priority inversion in a running SLOTH system. The figure shows execution traces of an OSEK OS application with a medium-priority interrupt (with its handler `ISR2`) occurring during the execution of the high-priority task `Task3`. In the commercial OSEK OS system, `Task3` is interrupted and `ISR2` is executed immediately—leading to rate-monotonic priority inversion (a). In SLOTH, since `Task3` runs at high interrupt priority, the medium-priority interrupt service routine `ISR2` is deferred until after `Task3` terminates—preventing rate-monotonic priority inversion (b).

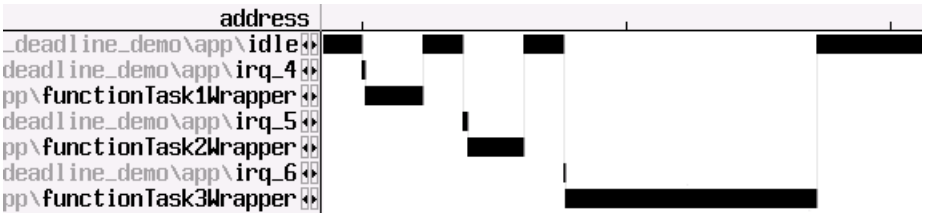
during the idle time of a dispatcher round, it keeps the system from sleeping, unnecessarily increasing its power draw. More importantly, if the deadline check interrupt is triggered during the execution of another application task, that task is unnecessarily interrupted, prolonged in its execution, and might miss its own deadline although it might be completely unrelated to the original task.

SLOTH ON TIME systems are designed in such a way that tasks enable their deadline cells in their prologues and disable them in their epilogues using a single memory-mapped store instruction (see Section 4.4.4). This way, deadline interrupts only occur if a task really misses its deadline, but not when it does not violate it, which is the common case.

To illustrate this, Figure 6.5 shows execution traces of an example application with three tasks and two deadlines, which are scheduled during the execution of `Task3`. In the commercial OSEKtime system, both deadline check interrupts issued to see if the corresponding task is still running lead to an interruption of `Task3` for 95 clock cycles each (see Figure 6.5(a)). This number multiplies by the number of deadlines to yield the overhead per dispatcher round in the commercial OSEKtime system. In SLOTH ON TIME, `Task3` is executed continuously (see Figure 6.5(b)). SLOTH ON TIME trades this advantage for a slightly increased task activation and termination time to manage the deadline cells (see evaluation in Section 6.3.3). This overhead, however, facilitates the real-time analysis of the



(a) Execution trace with deadline checks in the commercial OSEKtime system.



(b) Execution trace with deadline checks in SLOTH ON TIME.

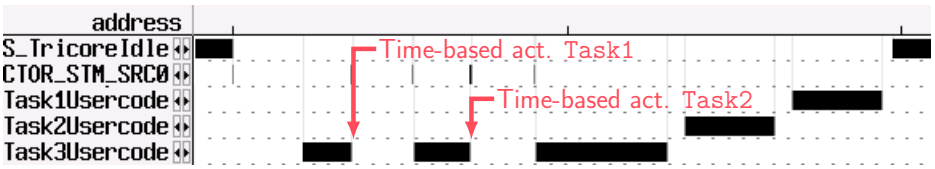
Figure 6.5: Prevention of deadline check interrupts in a running SLOTH ON TIME system. The figure shows execution traces of an OSEKtime application with three tasks and two deadlines, which are scheduled during the execution of `Task3`. In the commercial OSEKtime system, the deadline checks interrupt the execution of `Task3` (a); in SLOTH ON TIME, the deadline checks do not lead to an interruption of `Task3` (b).

application compared to deadline check interrupts and is easier to accommodate in the real-time schedule.

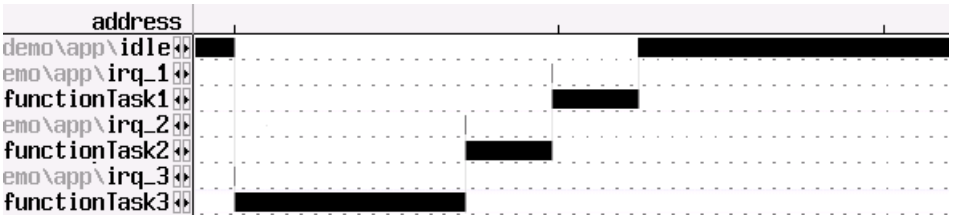
Note that the commercial operating system *could* be implemented in a way similar to SLOTH ON TIME to avoid additional interrupts; however, the overhead for its software logic would probably exceed the overhead that SLOTH ON TIME introduces (compare 12 cycles per deadline in SLOTH ON TIME to 95 cycles per deadline check interrupt in the commercial operating system).

6.5.3 Priority Inversion in AUTOSAR Systems

In AUTOSAR systems, event-triggered tasks with static priorities can be activated in a time-triggered manner using the schedule table abstraction, which can lead to situations of rate-monotonic priority inversion (see Figure 6.6). In the commercial AUTOSAR OS system, this situation can be observed when a low-priority task is activated by the timer while a high-priority task is running (see gaps in the execution of `Task3` in Figure 6.6(a)). The high-priority task is interrupted by the timer interrupt, whose handler then checks which task is to be activated and inserts this low-priority task into the ready queue. Thus, code is executed on behalf of a low-priority task while a high-priority task is running or ready to run, yielding a situation of rate-monotonic priority inversion. The priority inversion interruptions exhibited by the commercial AUTOSAR OS operating system are really heavy-weight: The corresponding handlers execute for 2,075 clock cycles



(a) Execution trace with time-based low-priority task activations in the commercial AUTOSAR system.



(b) Execution trace with time-based low-priority task activations in SLOTH ON TIME.

Figure 6.6: Prevention of indirect rate-monotonic priority inversion in a running SLOTH ON TIME system. The figure shows execution traces of an AUTOSAR application with three tasks with increasing priorities from 1 to 3; `Task1` and `Task2` are activated by a time-triggered schedule table during the execution of `Task3`. In the commercial AUTOSAR system, the activations of the low-priority tasks interrupt the high-priority task in a situation of indirect rate-monotonic priority inversion (a); in SLOTH ON TIME, `Task3` is executed continuously (b).

each. This can lead to serious deviations between the statically calculated WCET for a task and its actual run time, which is potentially prolonged by several of those low-priority interrupts.

In SLOTH ON TIME, those interrupts do not occur at all since the corresponding timer cell activates the task not by executing code on the main CPU but by setting the pending bit of the associated IRQ source, configured with the task priority. Since the high-priority task that is currently running runs at high *interrupt* priority, the activation does not lead to an interruption until the high-priority task blocks or terminates (see continuous execution of `Task3` in Figure 6.6(b)).

Thus, the SLOTH approach of running tasks as interrupt service routines in combination with the SLOTH ON TIME concept of using dedicated timer cells for time-dependent activations minimizes the number of interrupts in the system, facilitating real-time analyses. Interrupts only occur if an action needs to be taken by the operating system on behalf of the application. Traditional operating systems with a single system timer cannot avoid the described kind of unbounded priority inversion since they *have to* put the activated task into the ready queue (even if it has a low priority) and re-configure the timer to the next expiry point; this is an inherent design issue that SLOTH ON TIME overcomes. In SLOTH ON TIME, those problems are prevented by design; the timer hardware runs concurrently

to the main CPU and activates a task by setting the corresponding IRQ pending bit without having to interrupt the CPU.

6.6 Requirements on Other Hardware Resources

Besides the requirements on available data and code memory resources as well as CPU cycles to execute their system calls, the hardware-centric SLOTH operating systems also make use of other hardware resources to fulfill their function if the hardware platform can provide them. This includes interrupt resources in all SLOTH systems as well as timer resources in time-triggered SLOTH ON TIME systems; both of them depend on the application configuration as described in Section 5.4.

The SLOTH operating systems allocate one interrupt source per application task in the system—in addition to those interrupt sources connected to ISRs as specified by the application. Additionally, one interrupt priority slot including a corresponding vector table entry is reserved for each task, plus one priority slot and table entry for each resource in ECC1 SLEEPY SLOTH systems with resource support (see Section 4.3.6).

Time-based abstractions such as alarms, time-triggered dispatcher rounds, or schedule tables need interrupt sources connected to the hardware timer system plus one connected timer cell. The exact number depends on the number of alarms in the application and the number of operation points (e.g., task dispatch points or deadline check points) in the system dispatcher rounds and schedule tables—and on the fact whether the system uses multiplexing for some of those points or not.

Other than that, the SLOTH operating systems do not make use of special interrupts or traps. Note that all hardware resources allocated by SLOTH for the application can be configured by the application itself so that the allocation does not conflict with the resources the application needs for itself.

6.7 Evaluation Summary

The implementation of the SLOTH operating system concept is very concise in several different aspects. First, its source code size is very small; a complete OSEK BCC1 SLOTH operating system can be implemented in less than 200 lines of source code. Second, the compiled binary size is very small and scales with the demand of the application; typical data usage figures amount to less than 50 bytes and typical code size figures amount to less than 2 kilobytes including the compiler-provided start-up code. Third, the conciseness of the system call implementations is reflected in their performance and the latencies they inflict upon the application; commercial operating systems of the same class are outperformed by the SLOTH implementations by factors of up to 106. Additionally, SLOTH applications benefit from higher predictability due to fewer situations of

unbounded rate-monotonic priority inversion—the SLOTH concept can prevent those situations by design by unifying the priority space of tasks and interrupt service routines.

```
/* OS objects */
enum {
    Task1 = 1, /* ID and priority */
    ISR2 = 2, /* ID and priority */
    Task3 = 3, /* ID and priority */
    Task4 = 4, /* ID and priority */
    Resource1 = 3, /* ID and ceiling priority, shared by Task1, Task3 */
};
#define IRQ_ENTRY_TASK(name,number)\
void __attribute__((asection(".inttab", "a=32", "f=ax"))) irq_##number(void) { asm volatile(\
    "mov %d0, 0x0b80\n"\
    "mtcr $psw, %d0\n"\
    "isync\n" /* necessary, see AP32066 */\
    "st.a [+%a9], -4, %a8\n"\
    "mov.a %a8," #number "\n"\
    "svlcx\n"\
    "enable\n"\
    "j function" #name "\n"\
  ); }
#define IRQ_ENTRY_ISR(name,number)\
void __attribute__((asection(".inttab", "a=32", "f=ax"))) irq_##number(void) { asm volatile(\
    "mov %d0, 0x0b80\n"\
    "mtcr $psw, %d0\n"\
    "isync\n" /* necessary, see AP32066 */\
    "svlcx\n"\
    "enable\n"\
    "j function" #name "\n"\
  ); }
typedef uint8_t TaskType;
typedef uint8_t ResourceType;
typedef enum { RUNNING, WAITING, READY, SUSPENDED} TaskStateType;
enum {
    TASKMAXPRIO = 4, /* maximum priority of all tasks */
    MAXRESHELD = 1, /* maximum number of resources held at once by all tasks (conservative estimation: number of resources) */
    OSMAXPRIO = 4, /* maximum priority of all tasks and category 2 ISRs */
    RES_SCHEDULER = OSMAXPRIO,
    INVALID_TASK = 5, /* TASKMAXPRIO + 1 */
    NUMBEROFTASKS = 3, /* number of tasks */
};
#define TASK(TASKID) void function##TASKID(void)
TaskType resourceStack[MAXRESHELD];
uint32_t resourceStackPointer; /* points to next item to be used */
unsigned int suspendAllCounter;
unsigned int suspendOSCounter;
TaskType suspendOSPrevPrio;
register uint32_t tc1796CurrentTask asm("a8");
register void* tc1796CurrentTaskIDStackPointer asm("a9");
ALWAYS_INLINE void archSetCurrentTask(TaskType id) {
  tc1796CurrentTask = (uint32_t) id;
}
ALWAYS_INLINE void archSetCurPrio(uint8_t prio) {
  register ICR_t_nonv icr;
  icr.reg = _mfcr($icr);
  icr.bits.CCPN = prio;
  _mtcr($icr, icr.reg);
  _isync(); /* necessary, see AP32066 */
}
ALWAYS_INLINE uint8_t archGetCurPrio(void) {
  uint8_t prio;
  register ICR_t_nonv icr;
  icr.reg = _mfcr($icr); /* no _dsync() necessary (see ISA doc) */
  prio = icr.bits.CCPN;
  return prio;
}
/* generated vector table */
IRQ_ENTRIES_BEGIN
IRQ_ENTRY_TASK(Task1,1)
IRQ_ENTRY_ISR(ISR2,2)
IRQ_ENTRY_TASK(Task3,3)
IRQ_ENTRY_TASK(Task4,4)
IRQ_ENTRIES_END
INLINE void archInitIRQs(void) {
    /* Task1 */
    CPU_SRC0.bits.SRPN = 1; /* priority (and ID) */
    CPU_SRC0.bits.SRE = 1; /* enabled */
    CPU_SRC0.bits.SETR = 1; /* task is auto-started */
    /* ISR2, serial interface receive interrupt */
    ASC0_RSRC.bits.SRPN = 2; /* priority (and ID) */
    ASC0_RSRC.bits.SRE = 1; /* enabled */
    /* Task3 */
    CPU_SRC1.bits.SRPN = 3; /* priority (and ID) */
    CPU_SRC1.bits.SRE = 1; /* enabled */
    /* Task4 */
    CPU_SRC2.bits.SRPN = 4; /* priority (and ID) */
    CPU_SRC2.bits.SRE = 1; /* enabled */
}
```

Figure 6.7: Complete source code for a minimal OSEK BCC1 implementation of the SLOTH operating system for the Infineon TriCore TC1796. Note that the code has been manually inlined; in practice, it is modularized and separated between architecture/application-specific/independent parts (see Figure 5.2).

```
87  ALWAYS_INLINE void tc1796Setr(TaskType task) {
88      if (task == Task1) CPU_SRC0.bits.SETR = 1;
89      else if (task == Task3) CPU_SRC1.bits.SETR = 1;
90      else if (task == Task4) CPU_SRC2.bits.SETR = 1;
91  }
92  ALWAYS_INLINE unsigned char tc1796Srr(TaskType task) {
93      if (task == Task1) return CPU_SRC0.bits.SRR;
94      else if (task == Task3) return CPU_SRC1.bits.SRR;
95      else if (task == Task4) return CPU_SRC2.bits.SRR;
96  }
97  /* actual OSEK system calls */
98  INLINE void ActivateTask(TaskType id) {
99    _disable(); /* disable all task interrupts */
100   tc1796Setr(id); /* set service request flag */
101   tc1796Srr(id); /* read back to sync hardware and software */
102   /* worst case: wait for 2 arbitrations */
103   _nop(); _nop(); _nop(); _nop(); _nop(); _nop(); _nop();
104   _nop(); _nop(); _nop(); _nop(); _nop(); _nop(); _nop();
105   _enable(); /* enable all task interrupts; defined preemption point */
106   _isync();
107 }
108 INLINE void TerminateTask() {
109   asm volatile("rslcx");
110   /* restore task ID */
111   asm volatile("ld.a %a8, [%a9+], +4");
112   asm volatile("rfe"); /* implicit _enable(); */
113 }
114 INLINE void ChainTask(TaskType id) {
115   _disable(); /* or setCurPrio(OSMAXPRIO) or setCurPrio(id) to delay chained activation */
116   tc1796Setr(id); /* set service request flag */
117   tc1796Srr(id); /* read back bit to synchronize HW/SW */
118   /* wait for 2 arbitrations */
119   _nop(); _nop(); _nop(); _nop(); _nop(); _nop(); _nop();
120   _nop(); _nop(); _nop(); _nop(); _nop(); _nop(); _nop();
121
122   asm volatile("rslcx");
123   /* restore task ID */
124   asm volatile("ld.a %a8, [%a9+], +4");
125   asm volatile("rfe"); /* implicit _enable(); */
126 }
127 INLINE void Schedule() {
128   /* does not do anything on preemptive system */
129 }
130 INLINE void GetTaskID(TaskRefType id) {
131   *id = (TaskType) tc1796CurrentTask;
132 }
133 INLINE void EnableAllInterrupts() {
134   _enable();
135 }
136 INLINE void DisableAllInterrupts() {
137   _disable();
138 }
139 INLINE void ResumeAllInterrupts() {
140   suspendAllCounter--; /* not interruptible here since interrupts are already suspended */
141   if (suspendAllCounter == 0) _enable(); /* counter is guarded */
142 }
143 INLINE void SuspendAllInterrupts() {
144   _disable(); /* not interruptible after that */
145   suspendAllCounter++; /* counter is guarded */
146 }
147 INLINE void ResumeOSInterrupts() {
148   suspendOSCounter--; /* not interruptible here -> counter is guarded */
149   if (suspendOSCounter == 0) archSetCurPrio(suspendOSPrevPrio);
150 }
151 INLINE void SuspendOSInterrupts() {
152   /* intermediate store still needed if current task ID is known because it can have taken resources */
153   TaskType localPrevPrio = archGetCurPrio(); /* atomic read of current priority */
154   archSetCurPrio(OSMAXPRIO); /* not interruptible after that */
155   if (suspendOSCounter == 0) { /* only "one previous priority" necessary */
156     suspendOSPrevPrio = localPrevPrio;
157   }
158   suspendOSCounter++; /* counter is guarded */
159 }
160 INLINE void GetResource(ResourceType id) {
161   TaskType curPrio = archGetCPUPrio(); /* atomic read of current priority */
162   resourceStackPointer++; /* reserves a slot in the stack */
163   resourceStack[resourceStackPointer] = curPrio; /* access to reserved slot */
164   /* resource ceiling priority is stored in ID itself */
165   TaskType newPrio = (id > curPrio) ? id : curPrio; /* level must not decrease */
166   archSetCPUPrio(newPrio);
167 }
168 INLINE void ReleaseResource(ResourceType id) {
169   TaskType newPrio = resourceStack[resourceStackPointer]; /* get prio from stack */
170   resourceStackPointer--;
171   archSetCPUPrio(newPrio);
172 }
173 EXLINE int main(void) {
174   _enable(); /* start scheduling tasks, autostart tasks */
175   while(1) {
176     _nop(); /* idle at CPU prio 0 */
177   }
178   return 0;
179 }
```

Discussion 7

> All SLOTHs are built for life in the treetops. They spend nearly all of their time aloft, hanging from branches with a powerful grip aided by their long claws.
> [...]
> SLOTHs even sleep in trees, and they sleep a lot—some 15 to 20 hours every day. Even when awake they often remain motionless.
>
> *From* National Geographic Society Animal Facts*, 2013*

The SLOTH concept to design an embedded operating system tailored to the hardware platform below as presented in Chapter 4 yields several outstanding properties in the implementation of that concept. As suggested by the results of the evaluation in Chapter 6, many non-functional operating system properties are improved compared to traditional operating system designs; however, the SLOTH approach is not a general-purpose approach, since it is not ideally suited for every kind of target system.

This chapter discusses both the advantages of the proposed approach (see Section 7.1) as well as its applicability and limitations (see Section 7.2). Where appropriate, the previous chapters have shortly compared the SLOTH approach to related work; a comprehensive overview of related work is given in Section 7.3. An overview of the achieved goals and of how SLOTH addresses the problems stated in the motivation concludes this chapter (see Section 7.4).

7.1 Advantages of the SLOTH Approach

The goals stated in the motivation (see Chapter 1) for the design of an embedded real-time operating system target several aspects of system safety, operating system efficiency, and developer usability. This section explains how the SLOTH approach can improve on state-of-the-art operating systems in those respects and it compares the SLOTH approach to traditional operating system implementations with software schedulers.

7.1.1 System Safety

The key point of the SLOTH approach of implementing embedded real-time operating systems in a hardware-centric way is the positive effect it has on the safety properties of both the embedded application and of the operating system itself. This property is of high practical relevance—exemplified, for instance, by the recent conception of the ISO 26262 standard, which targets the functional safety in road vehicles.

Application Predictability

The run time properties of an application running on SLOTH are improved in a way that makes its execution more predictable compared to traditional operating systems with a software scheduler—therefore facilitating the real-time analysis of the embedded system in the planning phase. For one, all of the SLOTH system calls have a bounded and deterministic execution time as in any other real-time operating system. Other than that, the positive effect on the predictability of the SLOTH application execution is due to several factors.

Prevention of Rate-Monotonic Priority Inversion. First, the motivating situation of rate-monotonic priority inversion at run time—low-priority, hardware-managed ISRs can always interrupt the execution of high-priority, scheduler-managed tasks—can be prevented completely by the SLOTH operating systems since they implement a unified priority space for tasks and ISRs. Both regular event-triggered operation and time-based activations in event-triggered SLOTH systems do not exhibit this kind of priority inversion at run time (see evaluation in Section 6.5). This includes the prevention of *indirect* rate-monotonic priority inversion, since SLOTH executes everything *on behalf* of a certain task *at the priority level* of that task—such as a time-based activation of that very task. This feature in SLOTH enables the real-time engineer to provide a tight and predictable schedule of the control flows in the system because interferences are excluded by design, facilitating the real-time system analysis before run time.

Avoidance of Unnecessary IRQs. Additionally, SLOTH systems do not use IRQs in situations where those are unnecessary at run time. The commercial time-triggered operating system used in the evaluation, for instance, executes an interrupt handler for every deadline check specified in a dispatcher round, wheres SLOTH ON TIME will only issue an IRQ when the corresponding deadline is actually violated (see evaluation in Section 6.5.2). Again, this operating system property makes execution more predictable and facilitates real-time analyses on the system by the real-time engineer.

Reduced System Jitter. Furthermore, SLOTH systems exhibit less overall system jitter due to less jitter when dispatching control flows at run time compared

to traditional operating systems with software schedulers—the SLOTH operating systems therefore offer an increased system scheduling precision. Apart from reducing the number of situations of rate-monotonic priority inversion (which is one kind of *unbounded* priority inversion) at run time as discussed before, this is due to less need for explicit synchronization of the operating system compared to software-based operating systems and due to the fact that time-based operating system actions do not interrupt the main CPU.

Additionally, the SLOTH operating system has very few sections that are explicitly synchronized compared to traditional operating systems with software structures. This is because most SLOTH operating system data is implicitly encapsulated in hardware state, and this state is only accessible from software via elementary operations, which the software perceives as being atomic. Thus, if SLOTH needs to modify operating system data, in many cases, it can do so using an atomic operation—without the need for explicit synchronization. In contrast, software operating systems need to explicitly synchronize concurrent operating system accesses to software data structures via blocking or non-blocking synchronization constructs; the result is jitter in the task dispatch times. Compared to such explicit software operating system synchronization, SLOTH can therefore reduce both the number of the situations potentially causing jitter and the jitter *range*, since the critical sections in hardware are smaller and the synchronization in hardware is faster. In SLOTH, the only remaining jitter source is the arbitration time of the interrupt hardware; however, that jitter source also has to be accounted for in software-based systems when control flows are activated asynchronously by an interrupt.

Furthermore, in event-triggered systems with time-based activations, the timer interrupt handler needs to be synchronized with the operating system since it accesses the operating system ready queue. If the operating system is synchronized in a blocking way, this can lead to jitter in the task dispatch times when the timer interrupt handler is delayed by a task executing a synchronized system call. If the operating system is synchronized in a non-blocking way, this can lead to jitter in the execution of the system call by the interrupted task. The SLOTH ON TIME approach eliminates that jitter source since explicit operating system synchronization is not necessary—the ready queue is implemented implicitly in the hardware IRQ state and does not have to be synchronized in software.

That is why the real-time engineer can use the higher predictability of the SLOTH operating systems to provide a tighter and more predictable application schedule; low release jitter numbers are especially important for control applications with demanding reaction latency requirements. Additionally, the low-jitter properties of the SLOTH implementation are crucial to be able to apply theoretical models in practice, which often do not properly take operating system jitter into account [ZSR98].

Reduced Jitter Through Reduced Cache Load. Moreover, the predictable execution of the application is increased indirectly due to a reduced cache load caused by the SLOTH operating system compared to larger, software-based operating systems (see also Section 7.1.2). The very small amount of memory it occupies both in code memory (see Section 6.2.1) and especially in data memory (see Section 6.2.2) makes it use only a very small amount of the corresponding caches as well—a BCC1 SLOTH operating system remains almost stateless. This way, more cache lines can be used by the application and fewer application cache lines are replaced by the operating system during system calls, thereby reducing the possibility of periods of cache line thrashing, which cause execution jitter in the application. If the operating system fits into available scratchpad memory, it can even leave the complete cache to the application, making it as predictable as possible at run time; whether or not this is possible can be determined when linking the system. Overall, the SLOTH property of system conciseness can reduce caching effects caused by the operating system; it therefore increases the predictability of the whole system and facilitates the development of the application's real-time schedule with tightened WCETs.

Support for Mixed-Criticality Systems. The unique property of the unified priority space provided by the SLOTH operating systems also makes them a good match for mixed-criticality systems such as frequently used in avionics, for instance. Due to the hardware priorities enforced by the interrupt controller, critical parts of the systems are guaranteed to have priority over less critical parts at run time. This is exemplified in SLOTH by the possibility for mixed-mode operation, running a lower-priority event-triggered system in the background time of the higher-priority time-triggered system (see Section 4.4.6).

Handling of Interrupt Overload Situations. An additional and unique SLOTH advantage resulting from the unified priority space is the possibility to handle situations of interrupt overload in the system. Since ISRs are equal to tasks in SLOTH, SLOTH systems can transparently use advanced *task* scheduling mechanisms such as the sporadic server [Liu00] to also handle *interrupt* scheduling in order to limit IRQ processing. Thus, in SLOTH, advanced scheduling mechanisms can be applied to *all* event-triggered control flows—both tasks and ISRs—in an integrated way.

In SLOTH systems, the application developer can freely distribute the semantic control flow priorities among tasks and ISRs, which leads to situations when a lower-priority ISR is deferred due to high-priority task execution (for an example, see t_3 in Figure 4.1). If both the device and the IRQ controller can only buffer a single IRQ occurrence (either signaled in an edge-triggered or level-triggered way) and a second IRQ from the same source reaches the IRQ controller during the high-priority task execution, that second IRQ cannot be buffered and is lost. However, this is a direct consequence of the configured priorities; if such a

situation is to be prevented, the IRQ has to be given a higher priority to begin with, as output by a preceding real-time analysis that takes into account interrupt inter-arrival times. Since the SLOTH overhead for IRQ processing is very low, it contributes to being able to avoid interrupt overload situations, given a correct application configuration.

Operating System Verification and Maintenance

The SLOTH property of a small code base (see Section 6.4) in its concise implementation make it a viable candidate for verification—a property of utmost importance to many safety-critical real-time systems. The reduced amount of source code compared to traditional, software-based embedded operating systems in combination with reduced code *complexity* due to fewer operating system code paths leads to a lower probability for the occurrence of program errors and to reduced efforts for an engineer to verify the operating system for correct operation. This intuitive fact is also expressed by what Tanenbaum calls his "first law of software" [Tan07]: "Adding more code adds more bugs." Furthermore, the system conciseness allows for easier maintenance while managing and evolving the operating system with regard to possible requirement adaptations, since it is easier to review its source code and to find the appropriate points where to adapt it.

Additionally, the system's straight-forward *synchronization*—a major concern in all concurrent and asynchronous systems—is tremendously simplified and therefore easier to verify, because, in SLOTH, the adjustment of the current CPU priority level is the single measure needed for all kinds of blocking synchronization demands. This includes both demands by the application itself and demands internal to the system to keep its data structures from being corrupted by asynchronous control flows. In those software-based systems that are synchronized in a blocking way, on the other hand, the synchronization of the operating system is usually performed by disabling hardware interrupts, whereas applications are typically synchronized using software measures. In SLOTH, the application synchronization demands are satisfied using the CPU priority level, namely

- by raising it to the resource ceiling priority to acquire a resource (see also Section 4.2.3),
- by raising it to the highest level of all configured tasks to disable preemption in a critical section (see also Section 4.2.5),
- by raising it to the highest level of all category-2 ISRs to implement `SuspendOSInterrupts()` (see also Section 4.2.2),
- and by raising it to the highest level of *all* ISRs to implement `SuspendAllInterrupts()` (see also Section 4.2.2).

The system-internal demands to keep the operating system synchronized are also implemented by raising the CPU priority level, namely to the highest priority of all tasks and category-2 ISRs (both of which can access system data structures) configured in a given system (see also Section 5.5.5). Since the operating system itself has very few points where it *does* need explicit synchronization due to its low demand on software data structures (see evaluation in Section 6.2.2), an analysis for the actual verification is further facilitated.

7.1.2 Operating System Efficiency

Since the total efficiency of an embedded system greatly depends on the size and nature of its deployed software, the hardware-centric SLOTH operating systems can improve on it in several respects related to CPU efficiency, memory efficiency, and energy efficiency.

CPU Efficiency

The measurements of SLOTH system call latencies as well as of other system latencies, such as time-triggered activations, show that its system performance is superior to the commercial operating systems in *all* microbenchmarks (see Section 6.3). Independent of the class of the deployed system—for instance OSEK OS BCC1, ECC1, OSEKtime OS, or AUTOSAR OS—the SLOTH operating system yields a performance advantage over software-based operating systems that the real-time application will always benefit from, since it can assert lower response times to the user and use the lowered CPU utilization to include additional functionality in the gained slack time. The exact amount of gained slack time depends on the degree that the application makes use of the SLOTH operating system compared to the application code itself: The more system calls it issues and the more operation points it specifies in its dispatcher tables, the higher the experienced benefit will be for the application.

The benefits of improved latency and system call performance introduced by the SLOTH concept have a positive impact on the schedulability of tasks in the application. For preemptive static-priority systems, the system overhead can be respected by adding the dispatch and termination overhead to the WCET of each task [Liu00]—yielding higher schedulability for the low-overhead SLOTH systems. In SLOTH ON TIME systems, this is directly perceivable by comparing the idle times in the execution traces in SLOTH ON TIME and the commercial operating systems (see Figure 6.5 and Figure 6.6); the increased slack time can be used to include additional application functionality by either extending existing tasks or by introducing additional time-triggered tasks.

In application scenarios with highly loaded schedules, an implementation using traditional, software-based operating systems might not even be possible due to the operating system overhead, whereas the reduced overhead in SLOTH systems might make it feasible. In systems with time-based activations, for instance,

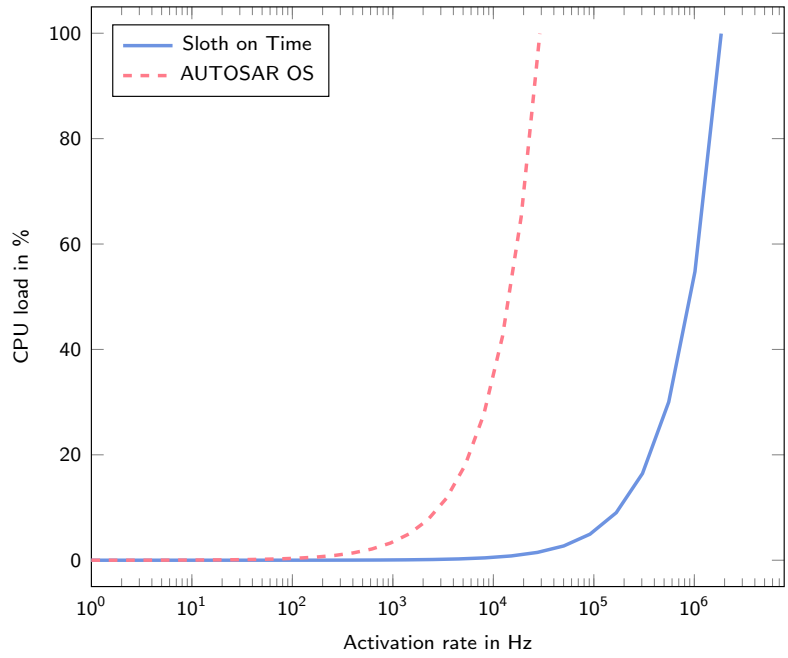


Figure 7.1: Plot of the activation rate of an event and the generated minimum CPU load. The plot shows the CPU load for systems running SLOTH ON TIME and the commercial AUTOSAR OS; note the logarithmic X axis. The maximum event activation rate of the AUTOSAR OS system is reached at 29 kHz, whereas SLOTH ON TIME supports activation rates of up to 1.8 MHz.

schedules with task release points that are very close together in time will cause problems in traditional operating systems, since the software scheduler will delay the second task activation through its dispatching overhead for the first task activation. By activating and dispatching in hardware, the very small overhead caused by SLOTH ON TIME can accommodate close activation points—as they occur when scheduling tasks with high activation frequencies, for instance. Taking into account the dispatching overheads caused by the operating systems as listed in the evaluation in Table 6.11, running on an Infineon TriCore with a CPU frequency of 50 MHz, SLOTH ON TIME supports a maximum dispatch frequency of 1.8 MHz of a single minimal task, whereas the commercial AUTOSAR operating system only supports 29 kHz, for example (see Figure 7.1). A certain class of real-time systems, such as high-speed packet switches, bears demands for high-resolution timing and scheduling; thus, inexpensive context switches such as those provided by SLOTH systems are needed to keep the operating system overhead as low as possible. Furthermore, the discussed activation rate differences can lead to the application developer being able to choose a smaller microcontroller derivative with less CPU power for running his real-time application, saving costs in the deployed product.

The number one reason for the improved performance in SLOTH systems is the re-location of operating system functionality from software to specialized hardware subsystems, hiding system latencies. Since the utilized interrupt controller and timer system build a kind of heterogeneous multi-core together with the main CPU, those subsystems can execute operating system functionality—namely control flow scheduling and dispatching—concurrently (see also Figure 4.16). In addition to executing functionality concurrently, the hardware can also execute the functionality *faster* compared to software; consider, for instance, the hardware-based dispatching of control flows compared to software-based context saving and restoring. Furthermore, the SLOTH operating systems are also preemptable at almost any point (a notable exception is the synchronization with the IRQ arbitration system; see Section 5.5.1); this increases the application responsiveness by keeping control flow activation latencies low. For all of these reasons, the latencies induced by the operating system in the SLOTH system calls can be kept very small, and the operating system executes more efficiently compared to traditional, software-based operating system implementations.

Although the main part of the good SLOTH performance comes from its adaptability to the hardware, another part comes from tailoring it to the needs of the application (both of which are enabled by SLOTH's flexible HAL). For instance, SLOTH incurs a lower overhead for the less powerful basic task abstraction compared to the extended task abstraction, whereas the commercial OSEK OS overhead remains almost the same for both kinds of abstractions (see Table 6.7 and Table 6.8). Additionally, the SLOTH operating system is tailored to the application through static analysis by the system generator and the compiler, which streamlines the operating system code for the application scenario, eliminating code paths known not to be reached at run time for the given application. In contrast, standard operating system implementations have lots of validity tests and case statements, since a large variety of use cases has to be handled by that operating system code (see also an analysis of general-purpose UNIX system call code in [PMI88]).

Besides the mentioned factors, on more sophisticated platforms with memory caches, the application will also yield a better performance on SLOTH since the low operating system memory footprint (see Section 6.2) also leads to a reduced cache load by the operating system. This way, more cache lines, both in the instruction and the data cache, can be occupied by the actual real-time application, speeding up its memory accesses and thereby its overall performance. This problem with larger operating system sizes has also been recognized by Liedtke [Lie95], who therefore started optimizing the L4 microkernel in this respect. If the microcontroller platform features fast (but small) scratchpad memory, the SLOTH operating system can be considered to be placed there directly due to to its conciseness.

Memory Efficiency

Besides reducing requirements on the CPU power of the hardware platform, the SLOTH design also reduces the operating system requirements on the employed data and code memory, as shown by the evaluation in Section 6.2. Since more operating system functionality is performed by the hardware platform compared to traditional implementations, the software part is reduced both in its code part and the encapsulated data.

Additionally, many variants of the SLOTH operating system can run on a single shared stack, reducing the stack part of the application's RAM demand. If no extended blocking tasks are configured by the application—that is, in BCC1 SLOTH systems, OSEKtime-like SLOTH ON TIME systems, and AUTOSAR-like SLOTH ON TIME systems with only basic tasks—a single stack is used by the whole system; the dispatcher uses the interrupt stack from the very beginning of the system start-up. In SLOTH systems with both basic and extended tasks, all basic tasks share a single stack. Additionally, since SLOTH system calls need less data compared to software-based operating systems, SLOTH tasks need less stack in the worst case and can therefore allocate less stack memory to begin with.

The reduced memory footprint is a very important property in the domain of deeply embedded systems, where single superfluous bytes in memory demand can lead to significant overall cost increase. If the deployment of a SLOTH operating system leads to the application developer being able to use a smaller microcontroller derivative, it can help save total costs of the embedded system. This is especially important in the automotive domain, where those costs multiply by tens of control units per vehicle, multiplied by possibly millions of vehicles produced and sold per year.

Since it has smaller system calls compared to software-based operating systems, SLOTH uses inlining of those system calls to a higher degree in order to achieve higher performance and stack usage efficiency; however, this design is subject to a trade-off decision. More inlined system calls and system code can also lead to a larger code size, requiring more ROM in the embedded device. However, RAM is about 8 to 16 times more expensive than ROM [DMT00], so the break-even point is not reached very quickly. Additionally, the SLOTH developer can configure the system to restrict inlining to reduce ROM size at the cost of increased stack usage and slightly decreased performance—inlining is a configurable property of the SLOTH system.

This trade-off is exemplified by the evaluation of the SLOTH code size scalability for the example applications in Section 6.2.1. In its standard variant, SLOTH "inlines" the task prologues by generating one statically tailored prologue per application task, which does not scale very well in terms of code size (see higher numbers in SLOTH code size ranges in Table 6.3), but which achieves very low task dispatch latencies. In its modified variant, SLOTH uses prologue *functions* with run time parameters to dynamically adapt to the dispatched task, which has

a positive effect on the code size (see lower numbers in SLOTH code size ranges in Table 6.3) but increases the task dispatch latencies by 5 to 15 clock cycles.

Energy Efficiency

Moreover, the reduced load induced by the operating system in SLOTH systems positively influences the energy consumption of embedded devices compared to systems using software-based operating systems. Since most embedded systems spend the majority of their time in sleep mode, the lower overhead introduced by the operating system has a direct impact on energy efficiency—and, therefore, on battery life, which is crucial in *mobile* embedded systems. System idle times, during which the device can be put to sleep mode, are reached faster and more often in SLOTH systems—for an example, compare the execution traces in Figure 6.5 or Figure 6.6.

7.1.3 Usability

Besides its effects on the non-functional properties of operating system efficiency and safety, the SLOTH design also simplifies application development. In a SLOTH system, the application developer is provided with an additional degree of freedom due to the unified control flow abstraction it offers. In contrast to the multitude of different control flow types offered by traditional embedded operating systems (see Table 2.1 in the problem analysis), a SLOTH application developer uses one single abstraction internally, reducing the development complexity.

Additional types of control flows that were introduced in traditional operating systems to offer more light-weight alternatives to the traditional tasks and ISRs (like OSEK callbacks and category-1 ISRs) are superfluous in SLOTH systems because the offered control flow type already has a very low overhead to begin with. In fact, SLOTH can offer OSEK tasks and category-2 ISRs at the price of an OSEK callback or category-1 ISR. In their implementations, SLOTH basic tasks and category-2 ISRs are almost the same, except that the former carry a task ID.

If the application developer needs to re-configure the properties of a control flow—including whether it is activated asynchronously by hardware or synchronously by software, whether it should be preemptable or not, or whether it should be run-to-completion or have blocking semantics—he can do that simply by adjusting the application configuration, without having to re-write parts of the application code to respect the new execution context and without having to alter the priority space design. The SLOTH system generator will then simply adapt the corresponding initialization and wrapper code if necessary. In traditional, software-based systems, such re-configurations can lead to significant needs for code re-factoring, since different control flow types have different properties with respect to possible operating system access, stack sharing, or prioritization, for instance (see also Table 2.1).

Additionally, the application developer is freed from the interaction constraints between different control flow types—such as different synchronization mechanisms for different involved control flow types. In SLOTH, synchronization between application control flows can always be performed via the resource abstraction in combination with a stack-based priority ceiling protocol—independent of the fact whether the involved control flow types are tasks or ISRs. In the OSEK OS specification, for instance, this participation of category-2 ISRs in the priority ceiling protocol is described as an *optional* (since complex) feature for the implementation, but it can be offered along the way by SLOTH without any additional programming or run time overhead. Communication and notification via events between tasks and ISRs is complicated to achieve in traditional systems with software-based operating systems, leading to considerable software overhead; in SLEEPY SLOTH, ISRs can simply wait for an event and be blocked if the application semantics needs them to, again providing additional freedom to the application developer.

Due to the unified priority space provided by SLOTH, the application developer can focus on designing meaningful control flows that reflect application semantics without being constrained by control flow properties. In a typical sensing application, for instance, the actual sensor data retrieval is performed in a short interrupt routine (triggered by either the sensor interrupt or a time-triggered interrupt), whereas the longer post-processing is performed in a separate task that is activated when exiting the ISR. Keeping ISRs short is a commonly taught and used application design pattern to minimize the effect of rate-monotonic priority inversion. Nevertheless, data has to be shared between the sensing ISR and the corresponding task, protected by appropriate synchronization mechanisms. In SLOTH, this semantically coherent routine can be performed in a single control flow—including both the sensing and the data post-processing in one longer, ISR-like control flow. By choosing an appropriate priority, other high-priority control flows in the system can be dispatched at any time in the SLOTH system. Additional flexibility stems from the fact that SLOTH control flows can lower their priority mid-program after their latency-critical part in order to buffer a burst after inserting a piece of data into a buffer, for instance.[1]

In traditional operating systems with software schedulers, it is impossible to have a task run at a higher priority than an ISR (since interrupt priorities are implicitly higher than task priorities), even though the application might demand it. SLOTH enables the application to freely distribute priorities among its control flows depending solely on its *semantic* requirements.

Finally, the application developer can choose to use newly combined properties offered by the SLOTH unified control flow abstraction—such as a blocking but hardware-activated control flow (i.e., a blocking ISR). In traditional, software-based systems, such a control flow type has to be emulated by executing a small

[1]The following lower-priority post-processing then has to be designed to be re-entrant and has to be respected by the application's stack calculation, of course.

ISR that directly activates an extended, blocking task; this workaround leads to additional overhead in the system and hinders real-time priority analyses.

7.2 Applicability and Limitations of the SLOTH Approach

The unconventional nature of the SLOTH operating system design makes it suitable for many kinds of embedded real-time systems, but it also bears limitations with respect to the offered operating system functionality and limitations depending on the underlying hardware platform.

7.2.1 Applicability of the SLOTH Operating System Design

Despite its simple design, SLOTH is suitable for the implementation of a wide range of real-time systems. For one, this includes event-triggered systems with fixed priorities. Additionally, the SLOTH ON TIME design allows for the efficient implementation of time-triggered real-time systems. The approach to implement run-to-completion tasks as ISRs, to implement extended tasks using ISRs with a tailored prologue, and to utilize timer cell arrays for time-triggered dispatching is generally applicable to any real-time operating system with an event-triggered or time-triggered interface.

However, the event-triggered design is applicable only to systems with static priorities. The SLOTH design is therefore suitable to implement the most well-known fixed-priority scheduling algorithms—like the rate-monotonic algorithm [LL73] and the deadline-monotonic algorithm [Liu00], which have been proven to be optimal for preemptive static-priority systems. Dynamic-priority algorithms—such as earliest deadline first [Liu00]—cannot be implemented with the presented SLOTH design; such systems need more comprehensive adaptations (see also considerations in Section 7.2.2 and in Section 8.2.2).

Legacy applications that are programmed using the API described in the OSEK OS, OSEKtime OS, or AUTOSAR OS standards can be used with SLOTH without modifications, since SLOTH directly implements those specifications as example interfaces. The existing application configuration, including task priorities and other properties (also defined by OSEK in its OIL OSEK implementation language [OSE04]), can also be used by the SLOTH generator to produce the configuration-dependent code (see also Section 5.1); a simple translator program can convert the OIL syntax to the SLOTH Perl hash syntax (for example, the line `'priority' => '1'` in Figure 5.1 is translated into `PRIORITY = 1;` in OIL syntax). Hence, no porting is needed for an OSEK or AUTOSAR OS application to benefit from SLOTH's advantages, and the application programmer can rely on the programming model and abstractions he is used to. Additionally, by having the SLOTH operating systems implement the OSEK and AUTOSAR OS standards, comparing them to other implementations is straight-forward since the benchmarking applications can be executed in an unmodified manner to evaluate operating system properties (see Chapter 6).

7.2.2 Operating System Functionality Limitations

Although the SLOTH approach is applicable to a wide class of real-time systems, its implementation cannot offer arbitrary functionality since it is constrained by the functionality of the hardware platform it is focused on. In this respect, SLOTH trades the generality of traditional software-based operating systems for predictability and efficiency in its execution to some extent. However, in some cases, there are ways around these constraints by careful software design on top of the hardware-centric SLOTH base if the features are requested by the application.

For instance, the missing blocking functionality in the original SLOTH design can be tolerated by many real-world applications, which avoid making use of that feature because of reasons of memory demand (e.g., stack sharing is hampered) and analyzability of the system behavior. If an application *does* need blocking tasks for reasons of flexibility, the SLEEPY SLOTH design can provide it by careful and optimized task prologue software (see Section 4.3).

Real-time systems with *dynamic* task priorities, which need to re-prioritize tasks at run time, are currently not supported by the SLOTH approach, since dynamic re-configuration of interrupt state at run time is usually very costly. The re-configuration of priorities in interrupt sources for re-prioritizing *new* task activations is inexpensive, but re-configuring the priorities of previously preempted tasks (i.e., SLOTH interrupt handlers) requires modifying stacked state, which is associated with higher software overhead. Thus, a careful software design is needed to address dynamic-priority systems (see also Section 8.2.2).

Another limitation of the current SLOTH implementation is that it does not support multiple task activations in combination with same-priority task groups. Application tasks with the same semantic priority can be mapped to adjacent but dedicated interrupt priority slots by the SLOTH generator, together with the task wrappers immediately raising the priority to the group priority. This way, activations of a task with the same semantic priority do not lead to a preemption of the running task, which might be needed for implicit synchronization between those tasks by the application. However, if the activation order has to be preserved for tasks of the same semantic priority (as demanded by the OSEK OS BCC2 and ECC2 conformance classes, for instance), a carefully designed software part is needed (see considerations in Section 8.2.1) since the hardware prioritizes using the distinctly assigned hardware priorities. If the dispatch order does not matter to the application, then the hardware priority assignment within one semantic priority group can be arbitrary. In most real-time systems, however, equal priorities will not be used, since they hinder a tight response time analysis. This is because, in the worst case, each task in the considered group can be delayed by all other tasks in the group, whereas with distinct priorities, only the lowest-priority task can be burdened by this high delay.

For advanced scheduling mechanisms, the system pre-condition is usually the ability to re-prioritize and suspend control flows in the operating system, which—as discussed before—is associated with a relatively high software overhead in

SLOTH. However, advanced real-time mechanisms such as the general priority ceiling protocol and aperiodic servers [Liu00] can still be implemented in SLOTH systems since they only require *occasional* re-prioritization of control flows by re-configuring the corresponding interrupt sources and modifying stacked state.

An additional limitation of the current implementation of the SLOTH operating system is that it does not include a system call that returns the current state of a given task (e.g., the OSEK system call `GetTaskState()`)—suspended, ready, running, or waiting. This is because the implementation would require a purely software-based design in one of two forms. In the first form, the system call would need to unwind the control flow stack to check if a task has been preempted (returning the ready state) or not (returning the suspended state), which can be very costly and which exhibits linear overhead. Alternatively, the task state can be tracked completely in software, using a software data field and creating overhead upon every dispatch operation and upon executing other system calls. Since the `GetTaskState()` system call is mainly used for debugging purposes, it is not important for production systems as targeted by SLOTH.

7.2.3 Hardware-Related Limitations

Since the SLOTH design is focused on the underlying hardware, which is where it draws its strengths from, it is also constrained by it in some respects—including the available hardware resources, the competition for them with the application, and the operating system porting effort. By enhancing the hardware platform, the applicability of the SLOTH approach can be further increased.

First, the feasibility of implementing a real-time application using SLOTH on a given hardware platform depends on its architecture and the available resources. As listed in Section 5.4, SLOTH poses certain requirements on the hardware platform for an ideal implementation of the operating system—such as to provide as many interrupt sources as there are application tasks in the system. If those resources are not available, software components can be used for multiplexing in some cases (see Section 4.4.8). Additionally, compromise solutions can be developed to reduce the effect of software multiplexing. For instance, if there are fewer timer cells available than task activation points, a dispatcher round can be divided into two halves so that there are only two re-configurations in software per round; the rest of the time, the timer array runs the dispatcher table autonomously, and no software overhead is incurred.

In the case of the number of available interrupt priorities, a theoretical foundation was built by Lehoczky and Sha to calculate the effect of an insufficient number of priority levels on the schedulability of a given task set [LS86]; additional considerations and calculations were performed by Liu [LL73] and Davis et al. [DMT00]. They find that the effect of insufficient priority levels on the utilization bound increases only very slowly and that 256 priority levels (as available on many platforms) are sufficient for a very high number of tasks (in the order of 100,000); the exact calculations can be found in [LS86].

As another example—in SLOTH ON TIME systems—since timer cells and connected interrupt sources are usually not freely configurable, the mapping of scheduling points to timer cells can be challenging for the developer of the hardware configuration model provided as an input to the generator (see Section 5.1.3). On the TC1796, for instance, restrictions apply that make it necessary to use two adjacent cells per activation; additionally, four cells are connected to a single interrupt source. Thus, on that platform, a second activation of the same task in a dispatcher round can be accommodated with minimal additional hardware resources. More than two activations will be subject to a trade-off decision, probably favoring a multiplexing implementation if cells become scarce (see Section 4.4.8). In general, the resources of the hardware platform and their interconnections will always be the limiting factor for the mapping of the application resources.

Second, for the implementation of their task abstractions, the SLOTH operating system implementations can only use those IRQ sources and timer cells that are not directly in use by the application. Additionally, many microcontrollers offer IRQ sources that can only be triggered from software and are thus ideally suited to be utilized by SLOTH. Concerning timer arrays, in practice, they are only used for control algorithms that bear latency requirements and activation rate requirements that are so tight that traditional real-time operating systems cannot fulfill them; by using the timer hardware directly, the application also becomes less portable. SLOTH ON TIME, on the other hand, still offers very low latencies, but hides its implementation beneath a platform-independent API and configuration, shielding the developer from porting the application from one hardware timer API to another. Thus, application developers from the respective domain are enabled to use low-latency time-triggered task abstractions as their units of decomposition instead of having to write low-level, platform-specific timer code.

Third, the hardware-centric nature of the SLOTH operating systems and their relying on efficient hardware mechanisms to perform the scheduling work for them imply a *relatively* high porting effort for its implementation since the overall size of the complete operating system is small and the hardware-independent software part of the operating system is also kept very small (see also Figure 3.2(b)). *Absolutely*, however, the hardware-specific parts—SLOTH's hardware abstraction layer in its static and flexible parts—are not larger than in traditional, software-based operating systems, and, thus, the efforts needed for porting SLOTH are not higher compared to those. Additionally, SLOTH has a clear hardware abstraction interface internally that specifies the parts that need to be mapped to the hardware platform (see also overview in Figure 5.2), resulting in a manageable porting effort, which is underlined by the availability of a short and clear hardware porting guide for SLOTH—including instructions on how to port the flexible part of the SLOTH HAL (see also Section 3.3.2). Since the whole SLOTH operating system is very small in its source and compiled binary form, it effectively builds a processor-specific layer for portable applications, which can rely on the platform-independent OSEK API. This way, the SLOTH approach performs

well in both non-functional properties of efficiency *and* portability (see also Section 2.3.2).

The design of the SLOTH system targets commodity off-the-shelf hardware platforms by making use of available IRQ subsystem and timer subsystem functionality. However, since SLOTH is limited by the underlying microcontroller as described above, hardware manufacturers could improve the suitability of their platforms for systems to be deployed using SLOTH by providing more corresponding resources on the chips. The required resources correspond to the requirements stated in Section 5.4 and include IRQ priorities, IRQ sources, and timer cells. These enhancements should not increase the complexity of the hardware logic itself since the logic has already been implemented and tested for fewer of those resources. The enhancements should not need a significant additional chip area either compared to *functional* additions in the microcontroller.

Functionally, hardware enhancements that can be made use of by SLOTH entail those that pose current limitations (see also Section 7.2.2) and those that can avoid less efficient software solutions. An example for the former includes hardware support for configurable scheduling and dispatching of same-priority interrupts, whereas the latter includes hardware support for multiple activations of an IRQ instead of providing only a single pending bit. These enhancements can increase the applicability of the SLOTH approach and make it even more efficient at run time.

Most current microcontroller platforms, however, *do* fulfill the requirements for SLOTH implementations as stated in Section 5.4. A recent study by EETimes confirms that more powerful 32-bit microcontrollers are spreading in embedded systems projects [UBM13]. Future embedded systems will therefore have even more features available in their underlying hardware platforms, ready to be used by SLOTH-like operating systems with high hardware abstraction interfaces. Current V850 E2M microcontroller cores by Renesas, for instance, feature a new subsystem called a timing supervision unit [Ren]. This unit operates as a kind of watchdog for supervising the execution of application tasks according to their statically specified limits. This supervision corresponds to the one proposed by the timing protection facility in AUTOSAR OS [AUT13] and includes supervision of task run times, task deadlines, run times of critical sections, run times of interrupt locks, and enforcement of IRQ inter-arrival rates. Once configured appropriately, the unit runs autonomously and concurrently, indicating an application malfunction using an exception. This new hardware subsystem is a perfect example of how hardware functionality can be integrated in a hardware-centric operating system such as SLOTH in order to provide operating system functionality at minimized overhead.

7.3 Related Work

The SLOTH approach of tailoring the operating system implementation to the peculiarities of commodity off-the-shelf hardware in order to optimize non-functional operating system properties is new and unique. This includes the bottom-up principle and hardware-driven way of an operating system design that is based on scrutinizing the hardware and its features and on pushing those up to the application.

The problems resulting from different types of control flows, however, have been discussed before (see Section 7.3.1)—as have some of the advantages resulting from tailoring operating systems to the underlying hardware (see Section 7.3.2). A lot of work concerning hardware support for operating system purposes has been done; in contrast to SLOTH, however, that work focuses on *customized* hardware using field-programmable gate arrays (FPGAs), for instance (see Section 7.3.3). Finally, optimizing operating system timer abstractions, as is the focus in SLOTH ON TIME, has been the focus of several pieces of operating system research (see Section 7.3.4).

7.3.1 Control Flows in Operating Systems

Several researchers have targeted the issue of different types of control flows in an operating system, the resulting problems, and how to manage them. They target the problem of rate-monotonic priority inversion and how to tackle it, the idea of running interrupt handlers in threads, and the execution semantics of control flows in operating systems and their efficiency.

Rate-Monotonic Priority Inversion

The main motivation for designing the SLOTH operating systems is mitigating the problem of rate-monotonic priority inversion, which occurs in systems with multiple control flow types and priority spaces (see Section 1.1). Leyva-del-Foyo et al. also target that problem—which jeopardizes predictability in real-time systems—and present an approach for integrated task and interrupt management [FMAN06a; FMAN12]. By hiding interrupts in the lowest level of the operating system, they convert IRQs into unblocking signals of corresponding interrupt service *tasks* (ISTs). These ISTs are initialized in a blocked state and are scheduled and dispatched in the same priority space as regular tasks once they have been unblocked by their IRQs. The disadvantage of the proposed model is that it increases the context switch overhead of the ISTs compared to regular ISRs, effectively offering hardware-activated ISRs at the higher overhead of software-activated tasks. SLOTH, on the other hand, does it the other way round and can offer *tasks* at the lower overhead of *ISRs*. The increased overhead is the reason why Leyva-del-Foyo et al. themselves propose hybrid systems with both regular ISRs and ISTs in the application to mitigate that problem to some extent.

Furthermore, they suggest to use a custom interrupt controller using an FPGA to implement their integrated scheme [FMA04] (see also other approaches using customized hardware in Section 7.3.3); additionally, they provide an example implementation on Intel x86 using a virtual interrupt controller that masks the interrupt in the 8259A PIC of the platform, introducing more additional overhead to scheduling, dispatching, and synchronization in the operating system.

To optimize the incurred overhead on Intel x86, Leyva-del-Foyo et al. extend their system by optimistic interrupt protection [SBBB93], which affects the implementation of the enter/leave protocol for critical sections in the operating system [FMAN06b]. When entering a critical section, the operating system only sets a software interrupt mask without the expensive port operations to change the hardware mask. Should the corresponding interrupts occur during the critical sections, they are deferred until after the critical section is left, which is when the operating system executes pending ISRs before resuming its computations; this decision is implemented in a small IRQ prologue and effectively preserves the interrupt locking semantics. This way, the hardware interrupt masking overhead can be avoided and can therefore optimize the common case—when the critical IRQs do not actually occur during the critical section—however, this increases the worst-case latency through the introduction of the prologue. The whole optimization is targeted toward architectures with external interrupt controllers or interrupt subsystems with relatively high configuration costs compared to the CPU frequency (e.g., superscalar or highly pipelined systems); on modern microcontrollers, which are targeted by SLOTH, both assumptions are generally not valid. In SLOTH, since entering a critical section by raising the CPU priority is even cheaper than setting a software mask, such optimizations are not needed. Additionally, in contrast to SLOTH, the proposed approach still yields periods of rate-monotonic priority inversion when an interrupt occurs during a semantically but not physically interrupts-disabled phase.

Dodiu et al. suggest a similar solution to the one proposed by Levya-del-Foyo et al.; by running tasks with masked interrupts, they effectively implement a unified priority space in the uC/OS-II operating system on ARM processors [DGG10]. The interrupt masks per task are generated a priori according to the unified priority space specified by the application. Upon task switches, the interrupt masks are adapted correspondingly, masking and thereby delaying semantically low-priority ISRs during the execution of the current task. However, this mechanism does not dispatch high-priority tasks when they are activated during the execution of a low-priority ISR; this behavior is needed to achieve full prevention of rate-monotonic priority inversion and would need an additional mechanism to be implemented in the scheduler. The model effectively runs tasks on interrupt level; in contrast to SLOTH, however, tasks are still scheduled in software with the associated overhead. Furthermore, additional overhead is introduced by having to maintain and re-program the interrupt masks upon all context switches.

In previous work together with colleagues, I investigated the suitability of using a co-processor for pre-processing interrupts to prevent rate-monotonic pri-

ority inversion [SHOPSPL09] (in a similar way, the PEACE operating system presented in Section 7.3.2 also uses one processor of a dual-processor system to act as an IRQ and scheduling co-processor). As in SLOTH, we used a specific hardware feature, namely the PCP co-processor of the TriCore microcontroller, to support the implementation of the CiAO operating system [LHSPSS09] in order to improve its non-functional properties. The co-processor pre-handles all interrupts by activating the corresponding task in the CiAO ready queue; it interrupts the execution on the main CPU only if a higher-priority task has become ready. This way, the single priority space is the software scheduler priority space, and rate-monotonic priority inversion is effectively prevented. However, as with the software-based solutions by Leyva-del-Foyo et al. discussed before, this property comes at the price of high software latencies for task and interrupt dispatching; additionally, the CiAO system has to synchronize the ready queue accesses by the main CPU and the co-processor, which further increases the worst-case latencies. SLOTH, on the other hand, prevents rate-monotonic priority inversion while *reducing* dispatching latencies for tasks and ISRs at the same time.

Zhang and West propose an approach to couple interrupt scheduling with the priority of the process that is associated to that interrupt to reduce priority inversion [ZW06]. In their modified Linux system, specific interrupts and their epilogues (named bottom halves in Linux) do not have a static priority but are scheduled depending on the receiving process. After executing the top half (i.e., the interrupt prologue), the scheduler uses heuristics to predict the associated process and its priority. Currently, the heuristics uses the highest priority of all threads waiting for the corresponding I/O device to make a scheduling decision for the bottom half. This way, rate-monotonic priority inversion can be reduced but not prevented as in SLOTH; additionally, the handling of the top halves remains unchanged and still incurs priority inversion.

Lee et al. extend the approach for UDP network interrupts [LLSSHS10]. In the bottom-half scheduler, UDP packets are analyzed for the receiving port, which can be mapped to a priority using a table in memory. The modified operating system then puts the bottom half into a queue according to its priority and only takes care of the packet if its process priority is higher than the currently running one. In contrast to SLOTH, however, this approach only targets rate-monotonic priority inversion for bottom-half scheduling and dispatching, but not for the top halves (i.e., prologues)—only *reducing* this kind of interference, but not *eliminating* it.

Manica et al. identify and investigate the problem of rate-monotonic priority inversion in Linux [MAP10] and Linux real-time versions using the PREEMPT_RT patch [RH07]. Linux interrupt threads, which are executed as kernel threads, mitigate the occurrence of situations of priority inversion; however, in mixed real-time systems with both real-time and non-real-time threads, fixed priorities do not allow for an acceptable trade-off between real-time guarantees and good quality of service in terms of throughput for hardware devices. Thus, Manica et al. propose to use reservation-based techniques such as the constant-bandwidth server [AB98; Liu00] for scheduling interrupt threads; they also de-

velop a model to correctly specify the corresponding reservation parameters for providing a good trade-off. The presented approach tackles trade-off problems in systems that, in addition to providing real-time guarantees, want to provide good throughput to non-real-time tasks. SLOTH, on the other hand, targets pure hard real-time systems and provides strict timing guarantees by preventing rate-monotonic priority inversion and by providing low and bounded system call latencies.

Regehr et al. investigate how systems distribute their control flows over multiple priority spaces, which they call execution environments [RRWPL03]. By using different execution environments such as category-1 ISRs, category-2 ISRs, and threads, such systems effectively implement and use a scheduling *hierarchy*, with root schedulers (e.g., the interrupt controller) having precedence over lower-level schedulers. This hierarchy causes problems when evolving an application and its control flows; moving functionality from a thread to a category-1 ISR, for instance, implies having to re-analyze the whole real-time system for synchronization constructs and corresponding blocking times. Regehr et al. show that by making this hierarchy explicit and by analyzing it, evolving a system by promoting or demoting code in the hierarchy is facilitated. SLOTH effectively implements a single priority space and thereby a completely flattened scheduling "hierarchy". As Regehr et al. point out, this is a property that makes SLOTH systems easier to analyze using real-time analysis techniques, and it makes it easy to promote or demote code by adjusting its priority without the need to change the used synchronization mechanism. Additionally, in a flat scheduling system such as SLOTH, rate-monotonic priority inversion cannot occur per se, since there are no higher-order control flow types that can lead to that type of priority inversion.

A completely different approach to bound the effect of interrupts on the real-time operation of the whole system is to use a completely time-triggered system such as originally proposed by Kopetz et al. [KDKMSSZ89]. In their Mars approach and operating system implementation, interrupts are disabled completely; instead, events are polled using a statically determined schedule, making the system completely deterministic, predictable, and resilient to interrupt overload by implicit flow control. In other forms, time-triggered systems do allow interrupts to trigger aperiodic or sporadic tasks to be dispatched in pre-allocated time slots in the time-triggered schedule table. SLOTH can be configured for event-triggered execution *or* time-triggered execution (see Section 4.4), depending on what is most suitable for the requirements of the real-time application.

Interrupt Threads

The notion of running interrupts as schedulable threads has been considered by several research projects with different approaches and implementation ideas. However, all of the proposed approaches have one common disadvantage: They increase the latency and overhead of the interrupt threads compared to regular interrupt handlers. Since threads, which are managed by the operating system,

incur significant software overhead compared to ISRs, the corresponding latency increases considerably—a drawback that the SLOTH approach prevents by running threads as ISRs if the hardware platform permits. Most of the systems presented in this section were not designed to prevent rate-monotonic priority inversion as SLOTH was, although most of them do reduce it due to their control flow designs.

In a notable publication entitled "Interrupts as Threads", Kleiman and Eykholt [KE95] show how the Solaris 2 operating system for desktop and server systems designs interrupt handlers as threads—going the opposite way compared to SLOTH. Their goal was to introduce blocking semantics into the ISR abstraction—a feature that SLEEPY SLOTH also offers. In Solaris, most interrupts are converted into threads with the corresponding thread properties so that they can use system services if they need to. This way, a single, unified synchronization model can be used throughout the operating system, and modular code can be designed oblivious to the context (interrupt or thread) that it is called in—both properties are also exhibited by SLOTH. However, the described Solaris system was not designed for real-time execution; it still exhibits rate-monotonic priority inversion when a high-priority regular thread becomes runnable but is kept from executing, since the operating system maintains the interrupt priority while executing an interrupt thread for synchronization reasons. In its *implementation*, Solaris 2 pre-allocates interrupt threads with stacks of their own but only makes them full-fledged threads that are schedulable by the system if the interrupt blocks, returning to the interrupted thread. This way, the blocking overhead for creating a full thread is only incurred on demand. However, for its interrupt threads, the Solaris 2 operating system incurs the *higher* overhead and latency of threads for its interrupts; SLOTH incurs the *lower* overhead and latency of interrupts for its threads.

Other systems that feature control flow abstractions similar to interrupt threads include Intel's early real-time operating systems such as iRMX [Rad] and iDCX 51 [Int], the L3 and L4 microkernels [LBBHRS91; Lie93; Lie95; Lie96; HHLSW97], MOOSE and its implementation variant AX [Sch86], and the network system by Mogul and Ramakrishnan [MR96]. These operating systems have a different focus compared to the SLOTH systems: They use interrupt threads for reasons of security and robustness, or to eliminate livelock problems, for instance. All of those systems schedule and dispatch interrupt threads in the operating system priority space, which reduces situations of rate-monotonic priority inversion in the running system as a side effect, since those interrupt threads become preemptable by higher-priority threads. However, they cannot prevent this type of priority inversion completely, since they execute a first-level interrupt handler before scheduling the corresponding interrupt thread; during that time, priority inversion is incurred.

Regehr et al. have also tried to implement interrupts as threads in radio processing in the TinyOS operating system to mitigate the problem of rate-monotonic priority inversion [RD05; RRWPL03]. However, they have found that this unac-

ceptably delays time-critical processing parts and induces persistent CPU overload since the additional scheduling and dispatching overhead makes the next IRQ arrive before the previous thread has been processed. Thus, high thread dispatching overhead can eliminate the possibility of moving IRQ processing to thread context for latency-sensitive real-time applications in TinyOS; in SLOTH, however, tasks are dispatched at ISR overhead, making them suitable for very latency-sensitive applications.

For the Composite component-based operating system, Parmer and West have a similar abstraction called an upcall, which is a thread associated with an interrupt [PW08]. Composite offers hierarchical user level schedulers, and an upcall has a priority that is respected by the corresponding scheduler in user space, which schedules the upcall in accordance with other thread priorities. Although the system generally prevents rate-monotonic priority inversion, it cannot do so completely: When an interrupt occurs, a short handler has to determine whether the associated upcall thread has a higher priority than the currently running thread. During that time, priority inversion is incurred; in SLOTH, on the other hand, not a single execution cycle is spent on low-priority interrupts.

The PREEMPT_RT patch tries to introduce real-time capabilities to the general-purpose Linux operating system through several measures, including adapted interrupt handling [McK05]. The patch makes interrupt handlers preemptable by running them in a process context, thereby eliminating rate-monotonic priority inversion for most of the ISR execution except for the first-level handler. The per-CPU timer interrupt plus user-specified exceptional ISRs, however, are still executed as hardware ISRs with high hardware priorities, incurring potentially unlimited periods of priority inversion, which SLOTH prevents entirely.

Execution Semantics

Several other run time systems have run-to-completion execution semantics that is comparable to the one offered by SLOTH. Some of those systems also target advanced blocking semantics as does SLEEPY SLOTH; they also focus on trying to keep the overhead low.

The Cilk run time system and thread scheduler was built for multi-threaded programming on parallel machines [BJKLRZ95; FLR98; Ran98] and defines continuation references to be passed among threads for communication and synchronization. Cilk threads, as SLOTH basic tasks, are run-to-completion C functions that cannot be suspended once they have been invoked. The reason for that is the same as in SLOTH: Run-to-completion threads leave the common stack empty when they complete; with suspension support, temporary values on the stack would have to be saved, or, as in SLEEPY SLOTH, separate stacks for each thread would have to be used. Cilk threads can explicitly allocate a cactus stack, which branches the stack at the spawn points between procedures; however, the branched parts of the stack cannot be used for the procedures to share data.

Newer resource-constrained systems such as TinyOS follow a similar approach, employing event-based tasks to reach high concurrency [HSWHCP00]. Hardware events can interrupt tasks, but tasks are executed atomically with respect to each other, dispatched by a FIFO scheduler. Since tasks have to be short and run to completion, a single stack can be used, which is especially important in the TinyOS target domain of wireless sensor networks. SLOTH tasks are more flexible since they allow preemption; they can still be executed on a single stack to save memory resources.

Protothreads [DSV05; DSVA06] target memory-constrained event-driven systems and provide a thread-like programming interface instead of having to specify explicit state machines in application code; as SLOTH tasks, they focus on low overhead. By employing local continuations, protothreads can be blocked and resumed in their execution; however, since protothreads are stateless and stackless and since the stack is re-wound upon blocking, automatic variables are discarded and therefore have to be saved manually by the application in a state object. Event-driven systems also use a single stack, and event handlers run to completion as in SLOTH; however, they cannot be preempted and, therefore, long-running computations have to be split up manually, which is where protothreads can facilitate application programming. By using a stack-based preemption pattern, SLOTH does not exhibit the problem targeted by protothreads, and *real* threads with stacks of their own as employed by SLEEPY SLOTH automatically save and restore automatic variables. Event-driven systems with protothreads, however, can further reduce the application stack usage by having to allocate only the *maximum* usage of all event handlers and not its *sum* as in OSEK BCC1 systems—all of this at the cost of impaired programmability. The protothreads approach can also be integrated in a run-to-completion SLOTH system to have blocking semantics with the mentioned restrictions; the approach is orthogonal to the SLOTH approach.

The REFLEX run time environment (real-time event flow executive) uses an event flow programming model to achieve implicit task synchronization for increased re-usability [WN06]. As in SLOTH, REFLEX tasks (called activities) share a single stack to reduce memory consumption and to achieve faster context switches; therefore, the tasks must not block but have to run to completion [WKSN08]. Like protothreads, activities are objects that can preserve state information without the need for a private stack. By employing static polymorphism mechanisms offered by the C++ language, REFLEX tasks can be specified in a scheduling-independent way; the actual scheduling policy (e.g., priority-based or time-triggered, and preemptive or non-preemptive) can then be flexibly set at compile time [WN07]. Thus, REFLEX tasks are universal to the programmer with respect to the scheduling policy; SLOTH tasks, in contrast, are universal with respect to scheduling, activation source, and execution semantics. Since REFLEX is designed to be used with an asynchronous data flow model, shared data is discouraged and not needed; thus, an interrupt can directly trigger a REFLEX activity. Blocking tasks with stacks of their own are not included in the REFLEX

design; SLEEPY SLOTH, on the other hand, extends the run-to-completion SLOTH model by this kind of task.

The stack resource policy (SRP) together with stack-based priority preemptive scheduling, both developed by Baker [Bak90; Bak91], propose an execution model for control flows that is used by SLOTH and also by OSEK in its conformance class BCC1. In order to reduce stack usage, SRP has tasks run to completion but allows preemption, thereby allowing for continuous allocation on one single stack instead of having to pre-allocate the maximum use. However, stack blocking can cause deadlocks and priority inversion; SRP therefore uses a stack-based priority ceiling protocol to synchronize potentially nested accesses to critical sections by tasks. This way, once a task has started execution, all requests to enter a critical section are granted immediately and without blocking; hence, no additional context switches are needed besides those for preemption, and the schedulability analysis is simplified. Baker notes, however, that a weakness of SRP is it that uses pessimistic assumptions about critical sections and can therefore keep other tasks from executing when this would not be needed compared to the regular priority ceiling protocol. SLOTH uses the very same mechanism as SRP to implement its resource abstraction and stack-like control flow scheduling with stack sharing, benefiting from Baker's techniques.

Draves et al. re-factor the Mach 3.0 kernel implementation to use continuations internally where possible [DBRD91]. A continuation saves its context explicitly to a dedicated memory area and specifies a function that is executed upon resumption; its stack can therefore be discarded while being blocked. The Mach 3.0 kernel effectively promotes continuations to first-class operating system abstractions to leverage optimization potential. SLOTH promotes *IRQs* and *interrupt handlers* to first-class kernel abstractions to also leverage optimizations; the main motivation for SLOTH, however, is to improve real-time properties such as to minimize situations of priority inversion. SLEEPY SLOTH *could* use continuations instead of dedicated stacks per extended task to save data space. However, since SLOTH embedded systems do not have a system call trap mechanism, there is no distinction between task stacks and a kernel stack, so stacks only have to be switched upon task switches but not upon every system call, which mitigates most of the advantage of continuations. Additionally, tasks would have to assess which data to save to their dedicated continuation memory areas upon every switch to another task. Thus, the task prologue model employed in SLEEPY SLOTH is a better fit for the given requirements.

The Fluke operating system [FHLMT99] can also be configured to use a single kernel stack by making kernel threads save their state explicitly to a dedicated memory area as with continuations. The SLOTH operating system effectively also features an interrupt execution model with a single stack for all user tasks as well as for the operating system. In SLEEPY SLOTH, blocking user tasks have a stack of their own, and the operating system executes on the stack of the current task since it is fully preemptable. However, due to the conciseness of the system calls

and—especially—their implementations in SLOTH, the operating system does not even use the stack on most platforms but can operate using only the register set.

The duality of synchronous threads and asynchronous ISRs leads to the need for interrupt synchronization in the operating system to keep operating system state from getting corrupted. Lohmann et al. give a good overview and analysis of techniques for operating system synchronization [LSSP05; LSSSP07]; in their CiAO operating system for embedded systems, the deployed technique can be configured at compile time. Simple systems usually employ hard synchronization by disabling IRQs during the critical section; this keeps the overhead low but increases the IRQ latency if only one critical section is rather long. More sophisticated operating systems such as Linux [Mau08; RC01], Windows [SR00], and PURE [SSPSS00] and its predecessors allow to divide interrupt handlers into time-critical prologue parts, which are executed on interrupt level, and epilogue parts, which are synchronized and executed on a special software operating system level that is also entered when a thread executes a system call. This divided-handler approach allows for low priorities for interrupt prologues while still allowing comprehensive, operating-system-accessing epilogue parts to be in sync with operating system state. Microkernel operating systems such as AX [Sch86] and L4Ka [Lie95], on the other hand, employ user-level driver threads; a generic IRQ handler in the kernel dispatches messages to the corresponding threads, which are even able to block. The Solaris interrupt threads described above are an optimized approach where the cost of running a full-fledged thread is only incurred on demand; this approach has also been adopted by Lohmann et al. for embedded systems [LSSSP07].

The execution model implemented by SLOTH combines the advantages of all approaches. The prologue–epilogue model can be used by the application by configuring category-1 ISRs and category-2 ISRs; the former are not synchronized with the operating system and therefore have a lower latency but are not allowed to call system services. SLEEPY SLOTH extended tasks are allowed to block during their execution but can be attached to an asynchronous interrupt source, effectively providing blocking ISRs. The SLOTH operating system implementation itself is synchronized using interrupt levels by having critical sections raise the CPU priority to the maximum of all tasks and ISRs that are allowed to access system services or by locking IRQs completely (see also Section 5.5.5); this decision is a latency–performance trade-off. The critical sections in the operating system, however, are all very concise, as is the operating system itself.

7.3.2 Operating System Tailoring to the Hardware

The SLOTH concept involves tailoring the operating system to the underlying hardware platform. By making use of architectural specialities, it can offer unprecedented non-functional properties—such as very low event latencies—to the real-time application running on top. This, however, reduces the portability of the lowest SLOTH layer to other hardware platforms. Previous work has also dealt

with the trade-off between performance and portability in operating systems that are optimized for a given hardware platform.

Back in 1983, Lampson published an influential paper with "Hints for Computer System Design" [Lam83], which includes general guidelines to be respected by good systems designers. He claims that the internal and external interfaces play a major role in determining the success of the resulting system, which is why many of the proposed rules target interface design. The most important interface rules, which are also respected by the SLOTH design, include to keep interfaces as simple as possible to incur reasonable cost; to generally be aware of the cost of interface design decisions; to opt for more basic but fast abstractions that can be combined to become more powerful; and, most importantly, to *not hide power*. When a lower layer offers a powerful abstraction, higher layers should not bury this power by generalizing; the principle of abstraction should not hide desirable properties in a system. This is exactly the core of the SLOTH operating system design principle: It makes use of special but powerful hardware abstractions and features for its operating system implementation—such as priority-based interrupt controllers and sophisticated timer arrays. The rest of the publication features lots of other good practices in systems building, many of which are relevant to and therefore respected by SLOTH, including using registers for fast access (see Section 5.5.3), and using static analysis to improve the system performance (see Section 5.1).

The Synthesis operating system design approach and its Synthetix follow-up project tackle the operating system trade-off between powerful features and efficient implementations by tailoring the operating system at run time, following the software engineering principle of frugality [PMI88; MP89; PABCCIKWZ95]. The code synthesizer component generates specialized system code for frequently executed system calls—such as a file-specific `read()` system call after opening that file, or tailored context switch procedures. This way, as SLOTH does, Synthesis and Synthetix generate an operating system that is tailored to the application and the available hardware features. In contrast to SLOTH, however, they perform this tailoring at *run time* instead of at *compile time*, which makes them adaptable depending on run time input, but which also produces higher run time latencies and jitter, jeopardizing deterministic execution. Thus, the Synthesis and Synthetix approaches developed for distributed systems are not applicable to embedded real-time systems, which are the focus of the SLOTH approach.

Similar to SLOTH, Brüning et al. also use hardware facilities to improve the performance of their PEACE operating system as perceived by the application by hiding communication latencies [BGSP94]. They target massively parallel message passing computers, whose dual-processor nodes they statically divide into an application processor and a communication co-processor part. PEACE uses clever separation of the point of communication and the point of synchronization on the application processor to hide the corresponding latency by having the communication processor act on the request before the application processor needs the result. Although Brüning et al. target parallel computers optimized

for performance and SLOTH targets embedded real-time systems optimized for determinism, both approaches hide latencies by using concurrent hardware subsystems for operating system purposes (see also Figure 4.16). Additionally, the PEACE approach increases the predictability on the application processor by handling all communication interrupts on the secondary processor.

The discussion about microkernel designs used to be dominated by its conceptual advantages such as modularity and isolation on the one hand, but also by their low performance due to increased local inter-process communication (IPC) compared to monolithic operating systems on the other hand [LBBHRS91; HHLSW97]. That is why follow-up work focused on improving costs for local IPC by clever kernel design and implementation on a given platform [Lie93; Lie95; Lie96]. One major factor in the resulting improvement by a factor of 20—a new level of efficiency—is the tailoring of kernel routines to the hardware particularities, similar to the way that SLOTH does it. The L3 kernel was thus improved to run more efficiently on the x86 platform by respecting the nature of its translation look-aside buffers (TLBs), its cache sizes and characteristics, its system call overhead, its segmented memory model, and its general-purpose registers. Liedtke recognized that a processor-specific implementation, using hand-coded assembly routines where necessary, as well the search for new design and implementation techniques such as the ones proposed by SLOTH, are required to achieve really high performance. In fact, Liedtke stated that, in his opinion, microkernels are *inherently* not portable, the same way as optimizing compilers, for instance. Portable operating systems cannot take advantage of specific hardware and cannot take precautions to avoid performance problems on a particular hardware platform; both restrictions imply restrictions on the provided non-functional properties of the resulting operating systems.

Liedtke also uses an optimization technique already proposed by Cheriton—register-based IPC [Che84]. For transfers of short messages, Cheriton's V microkernel uses the available general-purpose registers as an optimization instead of resorting to memory—a platform-specific optimization similar to SLOTH using a global register for holding the current task ID (see Section 5.5.3). Thus, as in SLOTH, performance-critical parts of the microkernel are written in a hardware-specific manner, possibly in assembly code. SLOTH goes even further by *inlining* its system calls into the application, which allows the compiler to perform a whole-program analysis, building the perfect base for optimizations by clever register allocation.

The exokernel architecture by Engler et al. reduces an operating system kernel to the very minimum functionality of securely multiplexing hardware resources [EKO95]. Applications can then bind to library operating systems, which provide specialized abstractions for different kinds of applications, such as database applications with specific file access patterns, for instance. Thus, the approach of using a minimal exokernel together with a customized library on top can yield a hardware-tailored system such as SLOTH. However, the target domain of an exokernel system is different from the SLOTH domain, since SLOTH systems

only run a single application and do not need protection boundaries—the main responsibility of an exokernel—in a typical requirements setup.

Bosch and Mullender designed a continuous-media server dictated by the measured characteristics of the underlying hardware platform [BM98]. Their Clockwise system software design offers applications a direct and efficient I/O data path to the hardware by exploiting special direct-memory-access capabilities. The measured performance of the resulting system is close to the raw hardware performance; by making use of hardware capabilities, SLOTH is also able to reach minimized latencies in its system calls.

The approach of application-oriented operating systems and the EPOS implementation by Fröhlich et al. use hardware mediators to specify a contract including both the interface and the semantics of a hardware component [Frö01; PF04]. A hardware mediator is a small and self-contained part of a traditional HAL designed to make the operating system itself portable, and it is selected by an EPOS operating system factory to match and adapt to the underlying hardware for best performance and flexibility. By using meta-programming techniques for the mediator and the rest of the operating system implementation, the different abstraction layers are compiled to inline functionality, introducing only very low overhead by adapting to the hardware as well as possible, similar to what SLOTH achieves with its design and implementation.

Handziski et al. present their hardware abstraction architecture to be used in software for wireless sensor nodes to strike a balance between the conflicting application requirements of portability and efficiency [HPHSWC05]. Since highly abstracted interfaces increase portability at the price of reduced efficiency (including performance and energy efficiency), their architecture allows applications to directly tap to platform-specific interfaces in efficiency-critical parts. Only the highest layer of abstraction, which they name hardware interface layer, is platform-independent with a typical interface for wireless sensor nodes and therefore either cuts off hardware features or needs to simulate them in software. SLOTH is different in that it offers a platform-independent API to the applications above, making *application* porting trivial. Only *internally*, SLOTH uses platform-specific functionality to implement this API by employing an internal hardware abstraction interface that is higher than in traditional operating systems.

The CiAO operating system, in whose development I was also involved in, also reaches good non-functional properties in its embedded operating system through high configurability [LHSPSS09]. CiAO uses a compositional approach to allow for tailoring the operating system to the application and the hardware platform; it employs design patterns based on aspect-oriented programming to modularize those tailoring concerns [LHSPS11]. Similar to SLOTH, the hardware-specific aspect code deploys clever platform optimizations, and the operating system inlines its system service implementations into the applications. However, since it is based on software data structures for its operating system operation, its footprint is not as concise as the one exhibited by SLOTH systems. Additionally, CiAO is susceptible to rate-monotonic priority inversion as any other software-

based operating system, except when using its PCP co-processor extension, which largely increases its latencies, however (see Section 7.3.1).

7.3.3 Operating System Support Through Customized Hardware

The optimization of non-functional properties of real-time operating systems through hardware use, especially for the core component of the operating system scheduler, has been the scope of several pieces of research. However, all of the following work uses *customized* hardware that has to be synthesized, and it moves operating system functionality to that custom hardware; SLOTH, on the other hand, targets commercial off-the-shelf hardware for its purposes. All of the approaches basically share the limitations by the hardware with SLOTH (see Section 7.2.3); however, since the hardware is customized, it can be adapted to accommodate for any shortcomings if enough chip area is available. The customization process also allows for arbitrary control flow semantics such as blocking threads to be implemented in hardware; SLEEPY SLOTH has to use short software prologues to implement blocking using the interrupt hardware (see Section 4.3.1).

The main problem with customized hardware is its high development cost and error proneness after production. Commodity off-the-shelf hardware, which the SLOTH approach is based on, is mass-produced at low cost and highly tested both on vendor side and in the field. Additionally, most customized hardware can only be run at lower clock frequencies compared to corresponding commodity off-the-shelf platforms. Thus, all of the following custom-hardware systems bear these disadvantages compared to SLOTH systems, which run on standard hardware.

The design of the original Spring operating system by Stankovic and Ramamritham targets distributed multi-processor systems and can therefore use dedicated system processors to off-load operating system scheduling overhead from application processors [SR89; MRSSZ90]. Additionally, IRQs are handled by the front-end I/O subsystem; an interrupt is treated like instantiating a new task in the system, effectively preventing rate-monotonic priority inversion as SLOTH does. In additional work, Stankovic et al. present a special co-processor, the Spring scheduling co-processor SSCoP, which exploits the parallelism inherent to scheduling algorithms to speed up the scheduling process [NRSWWBK93; BKNRSWW99].

Lindh, Stanischewski et al. describe their FASTCHART processor design, which features a dedicated real-time unit RTU [LS91b; LS91a; Sta93; AFLS96]. The main CPU offers real-time instructions that communicate with the RTU, which maintains ready, wait, and terminate queues and schedules tasks according to a static-priority algorithm. Task switches are performed by the RTU by exchanging the currently active register set by a second shadow version; this way, the latency for transferring the registers from and to memory are hidden from the main CPU, which experiences one CPU cycle for the context switch. One

restriction of the FASTCHART design is that the main CPU does not feature interrupts; by being restricted to polling external events, its performance is reduced in certain settings.

Silicon TRON by Nakano et al. implements basic real-time operating system functionalities in a peripheral chip, which improves performance by a factor of up to 50 and reduces the software operating system size to about one third compared to a pure software implementation [NUISI95; NKSI97]. Nakano et al. explicitly note that application interrupts are processed by Silicon TRON as direct task handlers, scheduling them in the same priority space as regular tasks; this way, the implementation prevents rate-monotonic priority inversion the same way SLOTH does, except that the latter runs on commodity instead of on custom hardware.

The concept of lazy receiver processing developed by Druschel and Banga incorporates several techniques to make the network subsystem of an operating system maintain its throughput and low latencies under high load [DB96]. Traditionally, the highest priority is given to the network interrupt handler, which can interrupt network post-processing, which itself has a higher priority than the user tasks awaiting the traffic; in highly loaded situations, this can lead to livelock due to rate-monotonic priority inversion [MR96]. In a hardware-oriented solution, the authors propose to use embedded CPUs present on high-end network adapters to filter packets based on the priority of the receiving process after de-multiplexing the destination socket, preventing priority inversion on the main CPU but not on the network adapter.

Dannowski and Härtig also target the problem of high interrupt loads generated by network traffic overload situations and propose to off-load scheduling of such interrupts [DH00]. To accommodate worst-case situations in real-time network systems, their approach shields the CPU from overload traffic by applying policies on RISC processors as available on network adapters themselves.

Hildebrandt et al. describe the implementation of a scheduling co-processor for the dynamic least-laxity-first algorithm to reduce the computational effort at run time [HGT99]. The co-processor is passive and needs to be triggered by the main CPU using a start signal, 34 cycles after which the result of the scheduling decision can be requested by the main CPU; the CPU still has to perform the task dispatch itself, however.

Saez et al. design a custom circuit to perform slack stealing scheduling, which induces a considerable scheduling overhead [SVCG99]. The circuit behaves similarly to a sophisticated interrupt controller and maintains task queues and slack times in hardware structures, avoiding overhead on the main CPU to increase the utilization of the real-time application.

The δ framework and its Atalanta operating system by Mooney III et al. provide an architecture for application-tailored hardware–software co-design [MB02; SBI02; AMITK03]. At compile time, δ supports the user in fine-grained partitioning of real-time system features between the system-on-chip hardware and the software part of the generated system, trading system speed-up for demands on number of gates and chip size. In additional work by

the same group, Kuacharoen et al. use re-configurable FPGA chips to implement a dynamically re-configurable hardware scheduler, whose scheduling algorithm can be changed at run time within 1.5 ms [KSM03]. This way, the scheduler remains flexible but can still eliminate most of the software overhead for task management.

The real-time task manager RTM by Kohout et al. is an on-chip peripheral that helps real-time operating systems perform their functions more efficiently [KGJ03]. The hardware module implements a task database and can be accessed via a memory-mapped interface by the operating system—to query it for the highest-priority ready task, for instance, in order to reduce the CPU utilization induced by the operating system.

Morton and Loucks present a system-on-chip design that moves the earliest-deadline-first scheduler into a co-processor named cs2 with the goal of speeding up the overall system execution by managing the ready list in hardware [ML04]. The cs2 co-processor is compiled and synthesized with a static number of application tasks and static task parameters, targeting similar real-time systems as SLOTH.

Regehr and Duongsaa also acknowledge the problem that interrupts are implicitly given higher priorities than threads and propose several solutions to prevent interrupt overload in embedded systems [RD05]. The proposed software solutions limit interrupt bursts by disabling and re-enabling the corresponding interrupt source using timers after configured inter-arrival times or after configured burst limits have been reached. The proposed hardware solution filters out interrupt signals directly on the IRQ lines, trading a small overhead in custom-chip area for zero overhead on the main processor.

Yamasaki et al. present their responsive multi-threaded processor (RMT), which integrates thread priorities in all of its functional units to resolve resource conflicts, yielding a scheduler in hardware [Yam05; YMI07]. Additionally, RMT provides a context cache [OH98], which can be swapped in and out in 4 clock cycles using special instructions to support fast scheduling and dispatching by the operating system. Interrupts can be assigned to waiting threads, which are immediately woken up upon the corresponding IRQ; however, in contrast to SLOTH, interrupt levels are in a different priority space compared to threads, which makes the system susceptible to rate-monotonic priority inversion.

Hthreads by Agron et al. is an operating system in hardware for the management of software and hardware threads to reduce system overhead and jitter while offering a uniform API for both types of threads [APAAKSBS06]. By moving thread management to hardware and by translating IRQs into thread scheduling requests, the operating system can execute in parallel to the main CPU, hiding the corresponding latencies and preventing rate-monotonic priority inversion.

HW-RTOS by Chandra et al. [CRL06] uses high-level synthesis tools to move selected functionalities of a traditional real-time operating system into hardware with the goal of achieving better performance. HW-RTOS moves the scheduling and communication between tasks into hardware, whereas the actual context

switching is still performed by the software on the CPU after receiving a context switch signal via an IRQ.

Lübbers and Platzner developed an operating system that supports regular software threads and hardware threads on re-configurable hardware to interoperate using a common programming model [LP07; LP09]. The ReconOS operating system allows data-parallel threads to be implemented in hardware while still being able to communicate and synchronize transparently with other hardware and software threads via standard system calls.

Hardware data structures by Bloom et al. represent the middle of a trade-off decision between flexibility and performance of a real-time scheduler [BPNS10]. By isolating task management *mechanisms* (i.e., the abstraction of hardware data structures) from task scheduling *policies* and implementing the former in hardware and the latter in software, the resulting system is both fast and flexible.

Olivier and Boukhobza propose to migrate the operating system timer handler into a configurable hardware component to reduce CPU load [OB12]. The FPGA component manages the system time and an array of tasks waiting to be woken up upon expiry of an alarm, which is done by interrupting the CPU only when re-scheduling is necessary; this leads to lower wake-up latencies for the corresponding tasks.

7.3.4 Optimized Timer Abstractions in Operating Systems

SLOTH ON TIME uses sophisticated hardware timer arrays to implement time-based operating system services more efficiently and predictably. Efficient and predictable implementation of timers has also been the focus of early as well as more recent operating system research; however, all discussed optimizations focus on the software level of timer management.

Varghese and Lauck discuss several implementation schemes for operating system timer facilities, which trade off different non-functional properties like insertion and interrupt latencies as well as memory requirements [VL87]. In their conclusion, they already propose special timer hardware to implement two of the schemes even more efficiently in order to relieve the main CPU even further. In their appendix, they propose a hardware counter chip that steps through the timer arrays and interrupts the host only if necessary; this is basically what SLOTH ON TIME does by assigning one application timer to a dedicated timer cell.

Zhou et al. analyze the problem of release jitter unpredictability caused by the implementation of operating system timers in three real-time operating systems [ZSR98]. In order to accommodate for rate-monotonic priority inversion and memory behavior of the operating system, they propose to respect those characteristics in the scheduling theory, presenting the extended RMTU model—rate-monotonic in the presence of timing unpredictability.

Soft timers as proposed by Aron and Druschel schedule events with very low overhead at the expense of probabilistic execution [AD00]. In addition to a low-frequency timer interrupt for bounding the execution delays, soft timers use trig-

ger states—such as system calls or exception handlers—to check the timer queue for expiries. At those trigger states, the operating system is already in supervisor mode and has already incurred the costs for polluting the caches and the TLB, making timer handler execution very inexpensive.

Fröhlich et al. compare timer implementations using one-shot hardware timers and periodic hardware timer ticks in terms of precision and interference due to incurred overhead [FGS11]. Periodic timer ticks incur overhead and interference with the application due to small phases of rate-monotonic priority inversion, and they are susceptible to rounding errors due to discrete tick lengths. One-shot timers do not exhibit those problems, but need to be re-programmed after each expiry, which is costly on most microcontroller architectures. The SLOTH ON TIME design, on the other hand, combines the best of both worlds; it only triggers interrupts when an event needs to be processed but does not have to re-program timers at run time.

Firm timers provide configurable non-functional properties in the implementation trade-off between timer accuracy and overhead [GAKSW02]. Internally, firm timers are one-shot timers—yielding good interrupt latency and power efficiency—but combine them with soft timers to avoid the interrupt context switch overhead as often as possible.

Peter et al. study the usage of timers in desktop operating systems and typical applications running on them [PBRBI08]. They find that the use of arbitrary timer values fixed at compile time leads to unnecessary wake-ups of the CPU during idle times and to significant jitter. In order to improve power consumption and responsiveness, they propose adaptive time-outs, such as used by the TCP network protocol, and higher-level, more explicit abstractions matching the need of the developers, such as nested timers and a precision parameter.

7.4 Goals Achieved and Problems Addressed

The motivation of this thesis (see Chapter 1) and its problem analysis (see Chapter 2) showed the way for the SLOTH approach and the SLOTH operating system design by stating the operating system goals and the problems in current state-of-the-art operating systems. This section re-visits those goals and problems and puts them into the context of the results of the SLOTH implementation and evaluation.

7.4.1 Goals Achieved

The motivation chapter states the goals for the development of a new type of embedded operating system in the context of this thesis (see Section 1.2). The SLOTH design and implementation presented in Chapter 4 and Chapter 5 achieve those goals; they are re-stated here, together with a short discussion of the corresponding SLOTH design elements.

Facilitate the choice between control flows

The goal was for the operating system to offer fewer and more universal abstractions to the application developer without sacrificing optimization potential. The SLOTH operating system achieves this goal by offering a single, universal control flow abstraction, which is internally implemented as an interrupt service routine. The universal SLOTH abstraction is independent

- of its type of activation—by software, by a timer, or by hardware periphery;
- of its preemption property—preemptable or non-preemptable;
- of its real-time execution mode—event-triggered or time-triggered;
- and of its execution semantics—run-to-completion or blocking.

Thus, all of the properties of an application control flow are a simple matter of configuration; its *implementation* and used synchronization mechanisms can remain the same. Nevertheless, a SLOTH application can be optimized to a very high extent, shown in the evaluation in Chapter 6.

Unify previously separate priority spaces

The goal was for the operating system to allow for the application developer to choose control flow priorities freely in order to prevent rate-monotonic priority inversion. The single SLOTH control flow abstraction enables the developer to do exactly this: By implementing all control flows as ISRs with IRQ priorities internally, the SLOTH operating system allows for arbitrary priority distributions among application control flows. This includes the possibility of giving higher priorities to software-activated control flows (i.e., task-like control flows) compared to hardware-activated control flows (i.e., ISR-like control flows). This way, both the direct type and the indirect type of rate-monotonic priority inversion are prevented by SLOTH *by design* (see evaluation in Section 6.5).

Expose advantageous non-functional properties of the hardware platform below

The goal was for the operating system to raise its system-internal hardware abstraction interface in order to be able to use peculiar and advantageous higher-level hardware features for the implementation of the application or of the operating system itself. Therefore, the SLOTH approach uses a flexible hardware abstraction layer that allows for the easy generation of hardware-tailored and application-tailored layer code, adapting in both dimensions—downward and upward. The SLOTH operating system achieves the goal by explicitly encouraging the exploitation of hardware particularities in its design for operating system purposes, which prototypically includes the utilization of sophisticated hardware interrupt controllers

and timer cell arrays (see Chapter 4). Nevertheless, SLOTH possesses a clear hardware abstraction interface to separate hardware-independent and hardware-specific parts (see also Figure 5.2) in order to facilitate its porting to other platforms (see Section 7.2.3).

Exhibit other favorable non-functional properties
The subgoal was for the operating system to exhibit good non-functional properties that are important in the embedded domain. The evaluation of the SLOTH operating systems shows that they do have low event latencies and fast and deterministic system calls (see Section 6.3) as well as low memory footprints in RAM and ROM (see Section 6.2).

7.4.2 Problems Addressed

The analysis in Chapter 2 revealed several problems pertaining to the design of state-of-the-art real-time operating systems. This section re-lists those problems in a grouped form, together with a short explanation of how SLOTH overcomes them.

1. **Priority inversion**
   Traditional operating systems exhibit several different forms of priority inversion, all of which are based on the fact that those systems have at least two separate priority spaces for tasks and ISRs. SLOTH features a single priority space—the interrupt priority space—for all types of control flows, which prevents all of the presented types of priority inversion. In SLOTH, control flows can be given a semantic priority that is absolute in the whole system and that is obeyed by the operating system at all times, including when it executes code *on behalf* of a control flow, such as a timer-based activation. This unique feature enables the application developer to freely distribute priorities in mixed-criticality systems and during the integration of real-time modules, for instance. Additionally, the application can use long-running ISRs with a low priority and use advanced scheduling techniques to shield the system from interrupt overload due to the unified priority space. Furthermore, the single priority space enables detailed real-time analyses as available from real-time theory, which often use that property as a simplification assumption.

2. **Synchronization problems**
   With traditional operating systems, the application has to deal with a multitude of different control flow types, and it has to carefully consider which synchronization mechanism to use between which of its control flows; in some scenarios, synchronization is even infeasible. Due to its universal control flow abstraction, SLOTH only has a single mechanism for synchronization between control flows, independent of the fact whether those are

hardware-activated or software-activated, for instance. The SLOTH operating system prototype offers a resource abstraction with a stack-based priority ceiling protocol, but other synchronization mechanisms are possible (e.g., see the integration of the priority inheritance protocol in Section 4.3.7). Through its single control flow synchronization mechanism, integration of module code is rendered trivial in SLOTH systems since the calling control flow context is always the same in SLOTH. Additionally, SLOTH application control flows can flexibly synchronize with other selected control flows in a fine-grained manner instead of having to resort to broadband synchronization mechanisms such as suspending *all* ISRs as in many traditional systems.

3. **Application development restrictions**
   In traditional operating systems, the development of applications is hindered and restricted by the availability of a multitude of control flows types with specific properties (see Table 2.1). SLOTH facilitates application development by offering a single control flow type that is configurable to the desired semantics as required by the application and not as implied by other control flow properties, combining previously infeasible property combinations (e.g., a hardware-activated ISR does not have to run to completion in SLOTH). The availability of a single control flow type also eliminates the need for artificial control flow ripping in the application for the single purpose of moving between different control flow types, and it eliminates the need for complicated analyses of access rights tables of the different control flow types.

4. **Application evolution inflexibilities**
   Traditional operating systems hinder smooth evolution paths for existing applications by restricting control flow flexibility. In SLOTH, application evolution is facilitated since control flows synchronize and communicate with a single mechanism. Additionally, due to the unification of control flow types, module code in SLOTH is always executed in the same context and can therefore be deployed and evolved flexibly.

Furthermore, the analysis in Chapter 2 identified the rigid and low location of the operating system hardware abstraction layer as one major cause for missed optimization potential, which is crucial in embedded systems. SLOTH therefore raises its hardware abstraction interface in order to be able to provide the application and the operating system implementation with beneficial hardware features although they might be particular to a certain hardware platform. To that extent, the SLOTH approach explicitly encourages the analysis of advantageous hardware features to be used in the operating system implementation. This way, the operating system and its flexible HAL can be tailored both to the application above and to the hardware platform below, yielding unprecedented non-functional properties in an embedded operating system.

To summarize, the SLOTH approach and design overcome the problems revealed in the analysis chapter by three unique properties in an embedded operating system:

1. SLOTH has a single, universal control flow abstraction.
2. SLOTH has a single priority space for all control flows.
3. SLOTH has a high hardware abstraction interface that exploits hardware particularities.

7.5 Discussion Summary

The SLOTH approach for a hardware-centric operating system design is different from traditional embedded operating system designs. By relying on hardware subsystems more comprehensively, the SLOTH operating system reaches higher efficiency levels, additional safety properties, and increased usability. The applicability of the operating system design is currently constrained to static-priority systems, and it is by design limited by the resources of the underlying hardware platform. The SLOTH design is new and unique; related pieces of research either overcome rate-monotonic priority inversion at the cost of efficiency loss or have to rely on custom-made hardware instead of being able to use off-the-shelf hardware as SLOTH does. SLOTH unifies control flow types and the corresponding priority spaces while exposing and exploiting advantageous properties of the hardware platform in order to prevent rate-monotonic priority inversion and to facilitate the development and evolution of real-time applications.

Conclusion and Outlook 8

> 6 Go to the ant, you sluggard;
> consider its ways and be wise!
> 7 It has no commander,
> no overseer or ruler,
> 8 yet it stores its provisions in summer
> and gathers its food at harvest.
>
> 9 How long will you lie there, you sluggard?
> When will you get up from your sleep?
> 10 A little sleep, a little slumber,
> a little folding of the hands to rest—
> 11 and poverty will come on you like a thief
> and scarcity like an armed man.
>
> *From the* Holy Bible*, proverbs 6:6–11, ca. 600 B.C.*

This thesis has presented the different facets of the SLOTH approach, which proposes to build embedded operating systems in a hardware-centric way. Section 8.1 summarizes the findings and contributions of this thesis, whereas Section 8.2 shows directions for on-going and future related research.

8.1 The SLOTH Approach and Operating System

The main contribution and foundation for the novel SLOTH approach for building statically configured embedded operating systems is its two-dimensional tailoring to both the application above and the hardware below, making it an ideally tailored operating system. By flexibly abstracting from the hardware at a higher level and by scrutinizing its peculiarities, SLOTH can make more comprehensive use of it and hide operating system latencies, pushing advantageous hardware features up to the application. In their implementations, by exploiting and instrumenting the interrupt subsystem and the timer subsystem, the event-triggered and time-triggered SLOTH real-time operating systems off-load operating system execution from the main CPU. The complementary goal—unifying the control flow abstraction for application tasks, ISRs, callbacks, and other control flow

types—is achieved in SLOTH by *internally* designating the interrupt handler as the universal control flow abstraction, which can by triggered asynchronously by the timer and periphery hardware, and which can by triggered synchronously by other control flows in software. *Externally*, applications are offered compatible legacy interfaces such as specified by the OSEK OS, OSEKtime OS, and AUTOSAR OS standards, for instance; thus, applications do not have to be ported to benefit from the SLOTH operating system design.

The main advantages of the SLOTH approach are threefold and apply to the optimization of the non-functional operating system properties of determinism, efficiency, and usability. The system determinism is improved by entirely preventing rate-monotonic priority inversion by design through the provided unified priority space; this feature makes the SLOTH design unique and facilitates the application of theoretical real-time models in practice, most of which do not take this kind of priority inversion into account. Additionally, the SLOTH approach also eliminates or mitigates a series of other real-time problems and restrictions related to the multitude of control flow types in traditional operating systems. The system efficiency (i.e., system call latencies and overhead, memory footprint for data and code, and—indirectly—energy efficiency) is improved by letting the hardware do most of the control flow scheduling and dispatching work, and by moving code and data from run time to compile time or initialization time as far as possible. SLOTH reaches control flow dispatch latencies of as few as 12 clock cycles, and it outperforms commercially used implementations of the standard interfaces by factors of up to 106. The system usability for the application developer is improved by providing a single control flow abstraction with unrestrained efficiency properties; the developer can therefore design, modularize, and evolve the real-time application with unrestricted assignment of priorities to control flows. The SLOTH approach abolishes the artificial distinction between tasks and ISRs and enables arbitrary interactions between hardware-triggered and software-triggered control flows, based solely on the requirements of the application.

In general, SLOTH exploits the fact that hardware is becoming ever more powerful and rich in features; this also includes *embedded* hardware. The results presented in this thesis should encourage operating system engineers to make better use of the hardware abstractions that a given platform offers. Especially in the domain of *embedded* operating systems, where a small footprint and efficient and deterministic execution are crucial, a small limitation in portability can often be traded for an improvement of those properties.

Since its conception, SLOTH and its variants have been used in several research groups from different universities. This includes research discussion groups at Washington University in St. Louis and at North Carolina State University; graduate level systems labs and courses at University of Minnesota, North Carolina State University, and Friedrich-Alexander-Universität Erlangen-Nürnberg; and an inter-disciplinary research project at Friedrich-Alexander-Universität Erlangen-Nürnberg. Additionally, a semi-conductor company is considering SLOTH to be

used by its customers for minimal working examples in its microcontroller application notes.

8.2 On-Going and Future Work

Recent, current, and planned work to enhance the SLOTH design targets the limitations stated in Section 7.2, which includes the possibility to include task groups at the same priority level (Section 8.2.1) and to implement dynamic-priority systems (Section 8.2.2). Additionally, it investigates research approaches on how to extend the scope of the SLOTH concept: its applicability to hybrid real-time systems (Section 8.2.3), its suitability for deployment on multi-core systems (Section 8.2.4), and its extension to provide control flow isolation (Section 8.2.5).

8.2.1 SLOTH with Same-Priority Task Groups

As described in Chapter 4, SLOTH and SLEEPY SLOTH prototypically implement the BCC1 and ECC1 classes of the OSEK OS standard, respectively; additionally, they offer multiple activations per task. The only feature that would be missing to provide the full functionality of the BCC2 and ECC2 classes of the OSEK OS standard is the ability to have multiple tasks with the same priority (see OSEK OS feature diagram in Figure 1.3). In the SLOTH system, such a feature is challenging to implement in combination with multiple task activations in an OSEK-compliant way since the standard prescribes that multiple activations of different tasks of the same priority have to be dispatched in the order of their activation—thus, this activation order has to be recorded in some manner.

This requested first-in-first-out behavior for different IRQ sources with the same priority is not offered by any interrupt hardware system and thus has to be carefully implemented in software in SLOTH, keeping the additional memory and latency overhead as low as possible in order not to jeopardize its design goals. Furthermore, the design has to respect the possibility of tasks being activated asynchronously by an alarm timer, which a software part of the SLOTH system has to enqueue correctly.

8.2.2 SLOTH for Dynamic-Priority Systems

As discussed in Section 7.2, the presented SLOTH design is limited to real-time systems with static-priority algorithms for their tasks. Dynamic-priority SLOTH systems—such as earliest-deadline-first (EDF) systems [Liu00]—would require re-prioritizing IRQ sources at run time, which needs to be considered when synchronizing with the IRQ arbitration system. Additionally, a proxy control flow would have to be deployed in order to assign the corresponding dynamic priority to the job of the task that has just been released. In general, the static nature of IRQ scheduling in interrupt systems is difficult to instrument for such run time re-configurations. Thus, an evaluation would have to show whether designing

dynamic-priority systems using the hardware-centric SLOTH approach is beneficial.

A non-intrusive approach to offering dynamic priorities in SLOTH systems is the insertion of an additional layer on top of the static-priority operating system. Diederichs et al. have shown how to implement a dynamic earliest-deadline-first scheduler on top of an OSEK-compatible system that assigns the task priorities statically according to the deadline-monotonic algorithm [DMSW08]. Their EDF plug-in layer between the application and the operating system implements a new delayed state of a task and activates a task in the underlying OSEK system according to its own scheduling data structure, which is a list of the tasks ordered by their deadlines. This way, tasks are delayed dynamically so that the one with the currently earliest deadline is always in the running state; the described plug-in could easily be integrated on top of SLOTH to provide the application with EDF scheduling as well.

8.2.3 SLOTH in Hybrid Real-Time Systems

The SLOTH approach targets embedded real-time systems; however, real-time systems are sometimes deployed in a hybrid setting together with a general-purpose operating system on the same hardware platform. In such a consolidated system, the real-time system takes care of the timely handling of periphery requests and communication, whereas the general-purpose system provides a sophisticated interface to the user.

The SLOTHFUL LINUX system design therefore uses the SLOTH operating system for the Intel x86 platform and deploys it beneath the general-purpose Linux operating system. In order to shield real-time execution from disturbances by Linux, SLOTHFUL LINUX uses a patch to *virtualize* interrupt handling in Linux: All interrupts are routed through an interrupt shield, which dispatches the IRQs either directly to the SLOTH real-time operating system or to the Linux operating system—in the latter case depending on a software bit that indicates whether or not Linux has its interrupts enabled. The Linux operating system is instrumented to modify this software bit whenever it attempts to modify the hardware interrupt enable bit, thereby making the execution of the real-time SLOTH operating system fully predictable. Thus, the implementation of the SLOTHFUL LINUX prototype can show that the SLOTH approach of interrupt-based real-time task dispatching and scheduling is well suited to be deployed in combination with an interrupt-virtualized general-purpose operating system to form a hybrid real-time system.

8.2.4 SLOTH for Multi-Core Systems

In the recent years, the advent of multi-core processing has also reached the embedded-systems market. Since multi-core processors are used mainly for consolidation purposes in that domain, a first migration step could introduce hard

task partitioning, which statically associates tasks with a core and schedules tasks only locally on a core depending on their static priorities. For core-local task synchronization, a standard resource abstraction can be used, whereas a spinlock abstraction needs to be introduced for mutual exclusion of tasks executing on different cores.

The SLOTH design can easily be enhanced to accommodate for multi-core systems by statically assigning each task to an interrupt source as usual, and by additionally configuring each interrupt source for execution on a dedicated core. This way, SLOTH task activations, which are implemented by pending the corresponding interrupt, are automatically routed to the correct core. Schedule tables, which encapsulate task activations, transparently work in the same way, routing time-based interrupt requests to the configured core. A prototype multi-core SLOTH implementation for the Intel x86 platform is already fully operational.

Further investigations will have to show if the SLOTH approach is also applicable to other domains beyond embedded real-time systems, which were originally targeted. This includes parallel systems, which also focus on low latencies and minimization of start-up times for dispatched control flows for efficiency reasons. The development of the multi-core design is the first basis for an approach that uses SLOTH as an execution platform for parallel systems.

8.2.5 Safety-Oriented SLOTH with Memory Protection

In order to provide a higher degree of safety, the SAFER SLOTH variant of the SLOTH operating system extends the implementation to provide configurable degrees of protection between SLOTH tasks, which are internally implemented as ISRs. SAFER SLOTH therefore uses link time arrangement of task-local and shared data to instrument the memory protection unit (MPU) of the hardware platform at initialization time and run time. Additionally, the SAFER SLOTH prototype provides several variants in the trade-off axis between the non-functional properties of low overhead and high safety that the application developer can choose from, depending on his application requirements.

In the first variant, which is applicable on hardware platforms that allow the MPU to be enabled in supervisor mode, SAFER SLOTH makes use of such a hardware peculiarity and runs tasks in supervisor mode as in the original SLOTH operating system, but it uses the MPU to isolate the tasks at run time. However, this variant does not prevent accidental re-configuration of the MPU *by the application*, which possesses the necessary supervisor privileges. To provide further isolation, SAFER SLOTH therefore offers a second protection variant, which executes tasks in user mode by revoking the supervisor mode rights in the task prologue. In this variant, the application uses a traditional system call trap, which dispatches the corresponding service from a system call table, returning to the application and to user mode after execution of the system call. Note that in the first variant, the safety of the system can only be guaranteed in combination with a static analysis of the application code for MPU re-configuration code—which

effectively moves parts of the safety enforcement from run time to compile time, following the SLOTH design philosophy.

Bibliography

[AB98] Luca Abeni and Giorgio Buttazzo. “Integrating Multimedia Applications in Hard Real-Time Systems”. In: *Proceedings of the 19th IEEE International Symposium on Real-Time Systems (RTSS ’98)*. 1998, pp. 4–13. DOI: 10.1109/REAL.1998.739726.

[AFLS96] Joakim Adomat, Johan Furunas, Lennart Lindh, and Johan Starner. “Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems”. In: *Proceedings of the 1996 Euromicro Workshop on Real-Time Systems*. 1996, pp. 164–168. DOI: 10.1109/EMWRTS.1996.557849.

[AHTBD02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. “Cooperative Task Management Without Manual Stack Management”. In: *Proceedings of the 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 289–302. ISBN: 1-880446-00-6.

[APAAKSBS06] Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. “Run-Time Services for Hybrid CPU/FPGA Systems on Chip”. In: *Proceedings of the 27th IEEE International Symposium on Real-Time Systems (RTSS ’06)*. 2006, pp. 3–12. DOI: 10.1109/RTSS.2006.45.

[AMITK03] Bilge E. S. Akgul, Vincent J. Mooney III, Henrik Thane, and Pramote Kuacharoen. “Hardware Support for Priority Inheritance”. In: *Proceedings of the 24th IEEE International Symposium on Real-Time Systems (RTSS ’03)*. Washington, DC, USA: IEEE Computer Society Press, 2003, p. 246. ISBN: 0-7695-2044-8.

[Infa] *AP32009, TC17x6/TC17x7 – Safe Cancellation of Service Requests*. Infineon Technologies AG. 81726 München, Germany, July 2008.

[ARM01] ARM Limited. *ARM7TDMI – Technical Reference Manual (Rev. 3)*. 2001.

[ARM10] ARM Limited. *ARMv7-M Architecture Reference Manual, Errata Markup*. 2010.

[AD00] Mohit Aron and Peter Druschel. "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing". In: *ACM Transactions on Computer Systems* 18.3 (Aug. 2000), pp. 197–228. ISSN: 0734-2071. DOI: 10.1145/354871.354872.

[AUT13] AUTOSAR. *Specification of Operating System (Version 5.1.0)*. Tech. rep. http://autosar.org/download/R4.1/AUTOSAR_SWS_OS.pdf. Automotive Open System Architecture GbR, Feb. 2013.

[Bak90] Theodore P. Baker. "A Stack-Based Resource Allocation Policy for Realtime Processes". In: *Proceedings of the 11th IEEE International Symposium on Real-Time Systems (RTSS '90)*. (Lake Buena Vista, FL, USA). Washington, DC, USA: IEEE Computer Society Press, Dec. 1990, pp. 191–200. ISBN: 0-8186-2112-5. DOI: 10.1109/REAL.1990.128747.

[Bak91] Theodore P. Baker. "Stack-Based Scheduling of Realtime Processes". In: *Real-Time Systems Journal* 3.1 (1991), pp. 67–99.

[BPNS10] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. "Real-Time Scheduling with Hardware Data Structures". In: *Proceedings of the Work-in-Progress Session of the 31st IEEE International Symposium on Real-Time Systems (RTSS '10)*. IEEE Computer Society Press, Dec. 2010.

[BJKLRZ95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*. Santa Barbara, California, USA: ACM Press, 1995, pp. 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958.

[BM98] Peter Bosch and Sape J. Mullender. "Don't Hide Power". In: *8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications (EWSCDA '98)*. ACM Press, 1998, pp. 152–157.

[BGSP94] Ulrich Brüning, Wolfgang K. Giloi, and Wolfgang Schröder-Preikschat. “Latency Hiding in Message-Passing Architectures”. In: *Proceedings of the 8th International Symposium on Parallel Processing (IPPS ’94)*. Washington, DC, USA: IEEE Computer Society Press, 1994, pp. 704–709. ISBN: 0-8186-5602-6.

[BKNRSWW99] Wayne P. Burleson, Jason Ko, Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles C. Weems. “The Spring Scheduling Coprocessor: A Scheduling Accelerator”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 38–47.

[CRL06] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. “Hardware/Software Partitioning of Operating Systems: A Behavioral Synthesis Approach”. In: *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI ’06)*. (Philadelphia, PA, USA). New York, NY, USA: ACM Press, 2006, pp. 324–329. ISBN: 1-59593-347-6. DOI: 10.1145/1127908.1127983.

[CMMS79] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. “Thoth, a Portable Real-Time Operating System”. In: *Communications of the ACM* 22.2 (Feb. 1979), pp. 105–115. ISSN: 0001-0782. DOI: 10.1145/359060.359074.

[Che84] David Ross Cheriton. “An Experiment Using Registers for Fast Message-Based Interprocess Communication”. In: *ACM SIGOPS Operating Systems Review* 18.4 (Oct. 1984), pp. 12–20.

[DH00] Uwe Dannowski and Hermann Härtig. “Policing Offloaded”. In: *Proceedings of the 6th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS ’00)*. Washington, DC, USA: IEEE Computer Society Press, 2000. ISBN: 0-7695-0713-1.

[DMT00] Robert Davis, Nick Merriam, and Nigel Tracey. “How Embedded Applications Using an RTOS Can Stay Within On-Chip Memory Limits”. In: *Proceedings of the Work in Progress and Industrial Experience Session of the 12th Euromicro Conference on Real-Time Systems (ECRTS-WiP ’00)*. 2000, pp. 43–50.

[DMSW08] Claas Diederichs, Ulrich Margull, Frank Slomka, and Gerhard Wirrer. “An Application-Based EDF Scheduler for OSEK/VDX”. In: *Design, Automation & Test in Europe Conference & Exhibition 2008 (DATE ’08)*. 2008, pp. 1045–1050. DOI: 10.1109/DATE.2008.4484819.

[DGG10] Eugen Dodiu, Vasile Gheorghita Gaitan, and Adrian Graur. "Improving Commercial RTOS Performance Using a Custom Interrupt Management Scheduling Policy". In: *Proceedings of the 2010 International Conference on Applied Computing (ACC '10)*. Timisoara, Romania: World Scientific, Engineering Academy, and Society (WSEAS), 2010, pp. 61–66. ISBN: 978-960-474-236-3.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems". In: *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*. (Pacific Grove, CA, USA). New York, NY, USA: ACM Press, Sept. 1991, pp. 122–136. ISBN: 0-89791-447-3. DOI: 10.1145/121132.121155.

[DB96] Peter Druschel and Gaurav Banga. "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems". In: *2nd Symposium on Operating System Design and Implementation (OSDI '96)*. Seattle, Washington, USA: ACM Press, 1996, pp. 261–275. ISBN: 1-880446-82-0. DOI: 10.1145/238721.238786.

[DSV05] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. "Using Protothreads for Sensor Node Programming". In: *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN '05)*. June 2005.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. Boulder, Colorado, USA, Nov. 2006.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM Press, 1995, pp. 251–266. DOI: 10.1145/224057.224076.

[FHLMT99] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. "Interface and Execution Models in the Fluke Kernel". In: *3rd Symposium on Operating System Design and Implementation (OSDI '99)*. Berkeley, CA, USA: USENIX Association, 1999, pp. 101–115. ISBN: 1-880446-39-1.

[FMAN12] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. “Integrated Task and Interrupt Management for Real-Time Systems”. In: *Transactions on Embedded Computing Systems* 11.2 (July 2012), 32:1–32:31. ISSN: 1539-9087. DOI: 10.1145/2220336.2220344.

[FMA04] Luis E. Leyva del Foyo and Pedro Mejia-Alvarez. “Custom Interrupt Management for Real-Time and Embedded System Kernels”. In: *Proceedings of the 2004 Embedded Real-Time Systems Implementation Workshop (ERTSI ’04)*. Washington, DC, USA: IEEE Computer Society Press, 2004.

[FMAN06a] 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 ’06)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2006, pp. 14–23. DOI: 10.1109/RTAS.2006.34.

[FMAN06b] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. “Predictable Interrupt Scheduling with Low Overhead for Real-Time Kernels”. In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA ’06)*. Washington, DC, USA: IEEE Computer Society Press, 2006, pp. 385–394. ISBN: 0-7695-2676-4. DOI: 10.1109/RTCSA.2006.51.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’98)*. Montreal, Quebec, Canada: ACM Press, 1998, pp. 212–223. ISBN: 0-89791-987-4. DOI: 10.1145/277650.277725.

[Frö01] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. GMD Research Series 17. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001.

[FGS11] Antônio Augusto Fröhlich, Giovani Gracioli, and João Felipe Santos. “Periodic Timers Revisited: The Real-Time Embedded System Perspective”. In: *Computers & Electrical Engineering* 37.3 (2011), pp. 365–375. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2011.03.003.

[GAKSW02] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. “Supporting Time-Sensitive Applications on a Commodity OS”. In: *5th Symposium on Operating System*

Design and Implementation (OSDI '02). (Boston, MA, USA). Berkeley, CA, USA: USENIX Association, Dec. 2002, pp. 165–180. DOI: 10.1145/844128.844144.

[HPHSWC05] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Corey Sharp, Adam Wolisz, and David Culler. "Flexible Hardware Abstraction for Wireless Sensor Networks". In: *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN '05)*. Istanbul, Turkey, Feb. 2005.

[HHLSW97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. "The Performance of μ-Kernel-Based Systems". In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. New York, NY, USA: ACM Press, Oct. 1997. DOI: 10.1145/269005.266660.

[Her91] Maurice Herlihy. "Wait-Free Synchronization". In: *ACM Transactions on Programming Languages and Systems* 13.1 (Jan. 1991), pp. 124–149. ISSN: 0164-0925. DOI: 10.1145/114005.102808.

[HGT99] Jens Hildebrandt, Frank Golatowski, and Dirk Timmermann. "Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems". In: *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS '99)*. 1999, pp. 208–215. DOI: 10.1109/EMRTS.1999.777467.

[HSWHCP00] Jason L. Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. "System Architecture Directions for Networked Sensors". In: *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, MA, USA, November 12–15, 2000*. New York, NY, USA: ACM Press, 2000, pp. 93–104.

[HDMSSPL12] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS". In: *Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS '12)*. (San Juan, Puerto Rico, Dec. 4–7, 2012). IEEE Computer Society Press, Dec. 2012, pp. 237–247. ISBN: 978-0-7695-4869-2. DOI: 10.1109/RTSS.2012.75.

[HLSSP09a] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. "Sloth: Let the Hardware Do the Work". In: *Proceedings of the Work-in-Progress Session of the 22nd*

ACM Symposium on Operating Systems Principles (SOSP '09). (Big Sky, MT, USA, Oct. 11–14, 2009). ACM Press, Oct. 2009.

[HLSSP09b] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. "Sloth: Threads as Interrupts". In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)*. (Washington, D.C., USA, Dec. 1–4, 2009). IEEE Computer Society Press, Dec. 2009, pp. 204–213. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.

[HLSP11] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads". In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS '11)*. (Vienna, Austria, Nov. 29–Dec. 2, 2011). IEEE Computer Society Press, Dec. 2011, pp. 67–77. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.14.

[Int] *iDCX 51 Distributed Control Executive User's Guide for Release 2*. Available at http://www.alfirin.net/flamer/bitbus/dcx51.zip. Intel Corporation. Apr. 1987.

[Rad] *Introducing the iRMX Operating Systems*. RadiSys Corporation. 5445 NE Dawson Creek Drive, Hillsboro, OR USA, Dec. 1999.

[JS93] Kevin Jeffay and Donald L. Stone. "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems". In: *Proceedings of the 14th IEEE International Symposium on Real-Time Systems (RTSS '93)*. IEEE Computer Society Press, Dec. 1993, pp. 212–221. ISBN: 0-8186-4480-X.

[KE95] Steve Kleiman and Joe Eykholt. "Interrupts as Threads". In: *ACM SIGOPS Operating Systems Review* 29.2 (Apr. 1995), pp. 21–26. ISSN: 0163-5980.

[KGJ03] Paul Kohout, Brinda Ganesh, and Bruce Jacob. "Hardware Support for Real-Time Operating Systems". In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*. Newport Beach, CA, USA: ACM Press, 2003, pp. 45–51. ISBN: 1-58113-742-7. DOI: 10.1145/944645.944656.

[KDKMSSZ89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach". In: *IEEE Micro* 9.1 (Jan. 1989), pp. 25–40. ISSN: 0272-1732. DOI: 10.1109/40.16792.

[KKK07] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. “Events Can Make Sense”. In: *Proceedings of the 2007 USENIX Annual Technical Conference*. (Santa Clara, CA, USA). Berkeley, CA, USA: USENIX Association, 2007, pp. 1–14. ISBN: 999-8888-77-6.

[KSM03] Pramote Kuacharoen, Mohamed Shalan, and Vincent John Mooney. “A Configurable Hardware Scheduler for Real-Time Systems”. In: *Proceedings of the Internal Conference on Engineering of Reconfigurable Systems and Algorithms*. CSREA Press, June 2003, pp. 95–101. ISBN: 1-932415-05-X.

[Lam83] Butler W. Lampson. “Hints for Computer System Design”. In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP ’83)*. (Bretton Woods, New Hampshire, USA). New York, NY, USA: ACM Press, 1983, pp. 33–48. ISBN: 0-89791-115-6. DOI: 10.1145/800217.806614.

[LR80] Butler W. Lampson and David D. Redell. “Experience with Processes and Monitors in Mesa”. In: *Communications of the ACM* 23.2 (Feb. 1980), pp. 105–117. ISSN: 0001-0782. DOI: 10.1145/358818.358824.

[LLSSHS10] Minsub Lee, Juyoung Lee, Andrii Shyshkalov, Jaevaek Seo, Intaek Hong, and Insik Shin. “On Interrupt Scheduling Based on Process Priority for Predictable Real-Time Behavior”. In: *ACM SIGBED Review* 7.1 (2010), p. 6.

[LS86] John P. Lehoczky and Lui Sha. “Performance of Real-Time Bus Scheduling Algorithms”. In: *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement, and Evaluation*. Raleigh, North Carolina, USA: ACM Press, 1986, pp. 44–53. ISBN: 0-89791-184-9. DOI: 10.1145/317499.317538.

[Lie93] Jochen Liedtke. “Improving IPC by Kernel Design”. In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP ’93)*. ACM Press, 1993. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168633.

[Lie95] Jochen Liedtke. “On μ-Kernel Construction”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95)*. ACM SIGOPS Operating Systems Review. ACM Press, Dec. 1995. DOI: 10.1145/224057.224075.

[Lie96] Jochen Liedtke. “Torward Real Microkernels”. In: *Communications of the ACM* 39.9 (Sept. 1996), pp. 70–77.

[LBBHRS91] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. “Two Years of Experience with a μ-Kernel Based OS”. In: *ACM SIGOPS Operating Systems Review* 25.2 (Apr. 1991), pp. 51–62. ISSN: 0163-5980. DOI: 10.1145/122120.122124.

[LS91a] Lennart Lindh and Frank Stanischewski. “FASTCHART – A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel”. In: *Proceedings of the 1991 Euromicro Workshop on Real-Time Systems*. June 1991, pp. 36–40. DOI: 10.1109/EMWRT.1991.144077.

[LS91b] Lennart Lindh and Frank Stanischewski. “FASTCHART – Idea and Implementation”. In: *Proceedings of the 1991 Internal Conference on Computer Design: VLSI in Computers and Processors*. 1991, pp. 401–404. DOI: 10.1109/ICCD.1991.139929.

[LL73] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *Journal of the ACM* 20.1 (1973), pp. 46–61. ISSN: 0004-5411.

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0-13-099651-3.

[LHSPS11] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. “Aspect-Aware Operating-System Development”. In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD ’11)*. (Porto de Galinhas, Brazil). New York, NY, USA: ACM Press, 2011, pp. 69–80. ISBN: 978-1-4503-0605-8. DOI: 10.1145/1960275.1960285.

[LHSPSS09] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. “CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems”. In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2009, pp. 215–228. ISBN: 978-1-931971-68-3.

[LSSP05] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. “On the Configuration of Non-Functional Properties in Operating System Product Lines”. In: *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’05)*. Chicago, IL, USA: Northeastern University, Boston (NU-CCIS-05-03), Mar. 2005, pp. 19–25.

[LSSSP07] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. “Interrupt Synchronization in the CiAO Operating System”. In: *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’07)*. (Vancouver, British Columbia, Canada). New York, NY, USA: ACM Press, 2007. ISBN: 1-59593-657-8. DOI: 10.1145/1233901.1233907.

[LP07] Enno Lübbers and Marco Platzner. “ReconOS: An RTOS Supporting Hard- and Software Threads”. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL ’07)*. 2007, pp. 441–446. DOI: 10.1109/FPL.2007.4380686.

[LP09] Enno Lübbers and Marco Platzner. “ReconOS: Multithreaded Programming for Reconfigurable Computers”. In: *ACM Transactions on Embedded Computing Systems* 9.1 (Oct. 2009), 8:1–8:33. ISSN: 1539-9087. DOI: 10.1145/1596532.1596540.

[MAP10] Nicola Manica, Luca Abeni, and Luigi Palopoli. “Reservation-Based Interrupt Scheduling”. In: *Proceedings of the 16th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS ’10)*. Washington, DC, USA: IEEE Computer Society Press, 2010, pp. 46–55. ISBN: 978-0-7695-4001-6. DOI: 10.1109/RTAS.2010.25.

[MP89] Henry Massalin and Calton Pu. “Threads and Input/Output in the Synthesis Kernel”. In: *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP ’89)*. New York, NY, USA: ACM Press, 1989, pp. 191–201. ISBN: 0-89791-338-8. DOI: 10.1145/74850.74869.

[Mau08] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, 2008. ISBN: 978-0-470-34343-2.

[McC04] Steve McConnell. *Code Complete*. 2nd ed. Microsoft Press, 2004. ISBN: 0-7356-1967-0.

[McK05] Paul McKenney. *A Realtime Preemption Overview*. http://lwn.net/Articles/146861/. Aug. 2005.

[MR96] Jeffrey C. Mogul and K. K. Ramakrishnan. “Eliminating Receive Livelock in an Interrupt-Driven Kernel”. In: *Proceedings of the 1996 USENIX Annual Technical Conference*. 1996, pp. 99–112.

[MRSSZ90] L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa. “Implementing a Predictable Real-Time Multiprocessor Kernel – The Spring Kernel”. In: *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*. Washington, DC, USA: IEEE Computer Society Press, 1990, pp. 20–26.

[MB02] Vincent J. Mooney and Douglas M. Blough. “A Hardware-Software Real-Time Operating System Framework for SoCs”. In: *IEEE Journal on Design and Test of Computers* 19.6 (2002), pp. 44–51. ISSN: 0740-7475. DOI: 10.1109/MDT.2002.1047743.

[ML04] Andrew Morton and Wayne M. Loucks. “A Hardware/Software Kernel for System on Chip Designs”. In: *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC ’04)*. (Nicosia, Cyprus). New York, NY, USA: ACM Press, 2004, pp. 869–875. ISBN: 1-58113-812-1. DOI: 10.1145/967900.968077.

[NKSI97] Takumi Nakano, Yoshiki Komatsudaira, Akichika Shiomi, and Masaharu Imai. “VLSI Implementation of a Real-Time Operating System”. In: *Proceedings of the 1997 Design Automation Conference*. 1997, pp. 679–680. DOI: 10.1109/ASPDAC.1997.600361.

[NUISI95] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. “Hardware Implementation of a Real-Time Operating System”. In: *Proceedings of the 12th TRON Project International Symposium (TRON ’95)*. Nov. 1995, pp. 34–42. DOI: 10.1109/TRON.1995.494740.

[NAS13] NASA. *Flight Software Branch (Code 582) OS Abstraction Layer Library*. Tech. rep. National Aeronautics and Space Administration (NASA), Goddard Space Flight Center, Jan. 2013.

[NDRTB07] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. *A SLOC Counting Standard*. Tech. rep. University of Southern California, 2007.

[NSP12] Vu Nguyen, Marilyn Sperka, and Ryan Pfeiffer. *UCC User Manual v. 2011.10*. http://sunset.usc.edu/research/CODECOUNT/download/2011/UCC_user_manual_v2011.10.pdf. Apr. 2012.

[NRSWWBK93] Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, Charles C. Weems, Wayne Burleson, and Jason Ko. “The Spring Scheduling Co-Processor: Design, Use, and Performance”. In: *Proceedings of the 14th IEEE International*

Symposium on Real-Time Systems (RTSS '93). 1993, pp. 106–111.

[OB12] Pierre Olivier and Jalil Boukhobza. "A Hardware Time Manager Implementation for the Xenomai Real-Time Kernel of Embedded Linux". In: *ACM SIGBED Review* 9.2 (June 2012), pp. 38–42. ISSN: 1551-3688. DOI: 10.1145/2318836.2318843.

[OH98] Amos R. Omondi and Michael Horne. "Performance of a Context Cache for a Multithreaded Pipeline". In: *Journal of Systems Architecture* 45.4 (1998), pp. 305–322.

[OSE01] OSEK/VDX Group. *Time-Triggered Operating System Specification 1.0*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf. OSEK/VDX Group, July 2001.

[OSE04] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf. OSEK/VDX Group, 2004.

[OSE05] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf. OSEK/VDX Group, Feb. 2005.

[PW08] Gabriel Parmer and Richard West. "Predictable Interrupt Management and Scheduling in the Composite Component-Based System". In: *Proceedings of the 29th IEEE International Symposium on Real-Time Systems (RTSS '08)*. Washington, DC, USA: IEEE Computer Society Press, 2008, pp. 232–243. ISBN: 978-0-7695-3477-0. DOI: 10.1109/RTSS.2008.13.

[PBRBI08] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. "30 Seconds Is Not Enough! A Study of Operating System Timer Usage". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*. (Glasgow, Scotland). New York, NY, USA: ACM Press, Mar. 2008, pp. 205–218. DOI: 10.1145/1357010.1352614.

[PF04] Fauze Valerio Polpeta and Antônio Augusto Fröhlich. "Hardware Mediators: A Portability Artifact for Component-Based Systems". In: *Embedded and Ubiquitous Computing*. Ed. by Laurence T. Yang, Minyi Guo, Guang R. Gao, and Niraj K. Jha. Vol. 3207. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 271–280. ISBN: 978-3-540-22906-3. DOI: 10.1007/978-3-540-30121-9_26.

[PABCCIKWZ95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. “Optimistic Incremental Specialization: Streamlining a Commercial Operating System”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95)*. New York, NY, USA: ACM Press, 1995, pp. 314–321. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224080.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. “The Synthesis Kernel”. In: *Computing Systems* 1.1 (1988), pp. 11–32.

[Ran98] Keith H. Randall. “Cilk: Efficient Multithreaded Computing”. PhD thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[RD05] John Regehr and Usit Duongsaa. “Preventing Interrupt Overload”. In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’05)*. New York, NY, USA: ACM Press, 2005, pp. 50–58. ISBN: 1-59593-018-3. DOI: 10.1145/1065910.1065918.

[RRWPL03] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. “Evolving Real-Time Systems Using Hierarchical Scheduling and Concurrency Analysis”. In: *Proceedings of the 24th IEEE International Symposium on Real-Time Systems (RTSS ’03)*. Washington, DC, USA: IEEE Computer Society Press, 2003. ISBN: 0-7695-2044-8.

[RH07] Steven Rostedt and Darren V. Hart. “Internals of the RT Patch”. In: *Proceedings of the 2007 Linux Symposium*. (Ottawa, Ontario, Canada). June 2007.

[RC01] Alexandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O’Reilly, 2001. ISBN: 0-596-00008-1.

[SVCG99] Sergio Saez, Joan Vila, Alfons Crespo, and Angel Garcia. “A Hardware Scheduler for Complex Real-Time Systems”. In: *Proceedings of the 1999 IEEE International Symposium on Industrial Electronics (ISIE ’99)*. Vol. 1. 1999, pp. 43–48. DOI: 10.1109/ISIE.1999.801754.

[SHOPSPL09] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Parallel, Hardware-Supported Interrupt Handling in an Event-Triggered Real-Time Operating System”. In: *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES ’09)*.

(Grenoble, France). New York, NY, USA: ACM Press, 2009, pp. 59–67. ISBN: 0-7695-2400-1. DOI: 10.1145/1629395.1629419.

[SSPSS00] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. "On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System". In: *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '00)*. (Newport Beach, CA, USA). IEEE Computer Society Press, Mar. 2000, pp. 270–277. DOI: 10.1109/ISORC.2000.839540.

[Sch86] Wolfgang Schröder. "A Family of UNIX-like Operating Systems — Use of Processes and the Message-Passing Concept in Structured Operating-System Design". In German. Dissertation. Technical University of Berlin, 1986.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. DOI: 10.1109/12.57058.

[SR00] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000 (3rd Edition)*. Microsoft Press, 2000. ISBN: 3-86063-630-8.

[SSL89] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. "Aperiodic Task Scheduling for Hard Real-Time Systems". In: *Real-Time Systems Journal* 1.1 (1989), pp. 27–60.

[Sta08] William Stallings. *Operating Systems. Internals and Design Principles*. 6th ed. Prentice Hall PTR, 2008. ISBN: 978-0136006329.

[Sta93] Frank Stanischewski. "FASTCHART – Performance, Benefits and Disadvantages of the Architecture". In: *Proceedings of the 1993 Euromicro Workshop on Real-Time Systems*. 1993, pp. 246–250. DOI: 10.1109/EMWRT.1993.639104.

[SR89] John A. Stankovic and Krithi Ramamritham. "The Spring Kernel: A New Paradigm for Real-time Operating Systems". In: *ACM SIGOPS Operating Systems Review* 27.3 (July 1989), pp. 54–71.

[Ste99] David B. Stewart. "Twenty-Five Most Common Mistakes with Real-Time Software Development". In: *Proceedings of the 1999 Embedded Systems Conference (ESC '99)*. 1999.

[SBBB93] Daniel Stodolsky, J. Bradley, Chen Brian, and N. Bershad. "Fast Interrupt Priority Management in Operating System Kernels". In: *Proceedings of the 2nd USENIX Symposium on Microkernels and Other Kernel Archtitectures (Micro '93)*. Berkeley, CA, USA: USENIX Association, 1993, pp. 105–110.

[SLS95] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments". In: *IEEE Transactions on Computers* 44.1 (Jan. 1995), pp. 73–91. ISSN: 0018-9340. DOI: 10.1109/12.368008.

[SBI02] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. *Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications*. Tech. rep. Georgia Institute of Technology, 2002.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd ed. Prentice Hall PTR, 2007. ISBN: 978-0136006633.

[Infb] *TC1796 User's Manual (V2.0)*. Infineon Technologies AG. St.-Martin-Str. 53, 81669 München, Germany, July 2007.

[Infc] *TriCore 1 User's Manual (V1.3.8), Volume 1: Core Architecture*. Infineon Technologies AG. 81726 München, Germany, Jan. 2008.

[UBM13] UBM Tech Electronics. *2013 Embedded Market Study*. Apr. 2013.

[Ren] *V850E2M User's Manual (Rev. 1.00): Architecture*. Renesas Electronics. Oct. 2012.

[VL87] George Varghese and Tony Lauck. "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility". In: *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*. (Austin, TX, USA). New York, NY, USA: ACM Press, 1987, pp. 25–38. ISBN: 0-89791-242-X. DOI: 10.1145/41457.37504.

[WKSN08] Karsten Walther, Reinhardt Karnapke, André Sieber, and Jörg Nolte. "Using Preemption in Event Driven Systems with a Single Stack". In: *Proceedings of the 2nd International Conference on Sensor Technologies and Applications (STA '08)*. Cap Esterel, France, 2008, pp. 384–390.

[WN06] Karsten Walther and Jörg Nolte. "Event-Flow and Synchronization in Single Threaded Systems". In: *Proceedings of the 1st GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES '06)*. 2006, pp. 1–8.

[WN07] Karsten Walther and Jörg Nolte. “A Flexible Scheduling Framework for Deeply Embedded Systems”. In: *Proceedings of the 4th IEEE International Symposium on Embedded Computing (ISEC ’07)*. 2007, pp. 784–791.

[Yam05] Nobuyuki Yamasaki. “Responsive Multithreaded Processor for Distributed Real-Time Systems”. In: *Journal of Robotics and Mechatronics* 17.2 (2005).

[YMI07] Nobuyuki Yamasaki, Ikuo Magaki, and Tsutomu Itou. “Prioritized SMT Architecture with IPC Control Method for Real-Time Processing”. In: *Proceedings of the 13th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS ’07)*. Apr. 2007, pp. 12–21. DOI: 10.1109/RTAS.2007.28.

[ZW06] Yuting Zhang and Richard West. “Process-Aware Interrupt Scheduling and Accounting”. In: *Proceedings of the 27th IEEE International Symposium on Real-Time Systems (RTSS ’06)*. 2006, pp. 191–201. DOI: 10.1109/RTSS.2006.37.

[ZSR98] Lei Zhou, Kang G. Shin, and Elke A. Rundensteiner. “Rate-Monotonic Scheduling in the Presence of Timing Unpredictability”. In: *Proceedings of the 4th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS ’98)*. 1998, pp. 22–27. DOI: 10.1109/RTTAS.1998.683184.

Kurzzusammenfassung

Software für eingebettete Systeme wird an die Anwendungsanforderungen angepasst, um die Kosten so niedrig wie möglich zu halten; dies betrifft insbesondere eingebettete *Betriebssysteme*, welche nicht zum Wert des Endprodukts beitragen. Aktuelle eingebettete Betriebssysteme passen sich an die darüberliegende Anwendung an, aber abstrahieren von der darunterliegenden Hardwareplattform, was vorteilhafte Hardwarebesonderheiten zur Optimierung nichtfunktionaler Eigenschaften des Systems ungenutzt lässt. Außerdem bieten sie dem Anwendungsprogrammierer eine Vielzahl an Kontrollflusstypen an, was zu diversen schweren Einschränkungen und Problemen bei der Anwendungsentwicklung und in der Echtzeitausführung des Systems führt – zum Beispiel, dass hochpriore Tasks jederzeit durch niedrigpriore Interrupt-Handler unterbrochen werden können.

Der Entwurf des SLOTH-Betriebssystems für ereignisgesteuerte und zeitgesteuerte eingebettete Echtzeitsysteme, welcher in dieser Arbeit vorgestellt wird, ist von besonderer Art, da er einen hardwarezentrischen Ansatz für die Einplanungs-, Einlastungs- und für zeitbasierte Dienste einsetzt und spezifische Hardwareeigenschaften ausnutzt – mit dem Ziel, die nichtfunktionalen Eigenschaften des Betriebssystems und der Anwendung zu optimieren. In seiner Implementierung weist SLOTH jedem Task eine Interruptquelle mit einer entsprechend konfigurierten Priorität zu und bildet Softwareaktivierungen dieses Tasks auf das Setzen des Request-Bits der Interruptquelle ab. Auf diese Weise lässt SLOTH das Hardware-Interrupt-Subsystem die Einplanungsentscheidung treffen und den entsprechenden Interrupt-Handler einlasten, welcher direkt die Taskfunktion der Anwendung ausführt. Zeitgesteuerte Einlastungstabellen werden in Arrays von Hardware-Timer-Zellen eingebettet, welche von SLOTH bei der Initialisierung mit den entsprechenden Zeitparametern vorkonfiguriert werden, um das Hardware-Timer-Subsystem die Einlastungsrunden zur Laufzeit autonom ausführen zu lassen.

Die Evaluation der Implementierung des SLOTH-Betriebssystems zeigt, dass der leichtgewichtige Entwurf die sicherheitsrelevanten Probleme und Einschränkungen bei der Echtzeitausführung der Anwendung reduziert oder sogar eliminiert, indem er eine einheitliche Kontrollflussabstraktion in einem vereinigten Prioritätenraum anbietet, was die künstliche Unterscheidung zwischen Tasks und Interrupt-Handlern beseitigt. Außerdem weisen die nichtfunktionalen Eigenschaften ein neues Niveau an Effizienz auf, indem SLOTH Latenzen in Hardware-Subsystemen versteckt: Ein vollständiges SLOTH-Betriebssystem kann auf weniger

als 200 Zeilen Code angepasst werden, auf weniger als 500 Bytes im Codespeicher und 8 Bytes im Datenspeicher kompiliert werden, und er kann Tasks innerhalb von 12 Taktzyklen einplanen und einlasten – mit Speed-Up-Faktoren von bis zu 106 im Vergleich zu kommerziellen Betriebssystemen. Da SLOTH prototypisch die Standards OSEK OS, OSEKtime OS und AUTOSAR OS implementiert, welche von der Automobilindustrie entwickelt wurden, und da das Betriebssystem auf Standard-Hardware läuft, ist der Ansatz anwendbar auf eine breite Menge an eingebetteten Echtzeitsystemen.