Efficient Time-Triggered Execution in an Interrupt-Driven Real-Time Operating System

Diplomarbeit im Fach Informatik

vorgelegt von

Daniel Danner

geboren am 18. März 1986 in Erlangen

Angefertigt am

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

Betreuer:

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

Beginn der Arbeit: 01.01.2012
Abgabe der Arbeit: 02.07.2012

Erklärung

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

Erlangen, 02.07.2012

Abstract

Traditional time-triggered real-time operating systems are usually implemented by multiplexing the predefined schedule onto a single hardware timer. At each hardware timer interrupt, the planned task is looked up in the static table, the timer is reprogrammed to the subsequent activation point and the task is dispatched. The overhead of this software effort leads to a considerable latency between interrupting the CPU and executing the actual task function.

This thesis explores the possibility of minimizing the overhead by making use of flexible timer cells as they are available in large arrays on modern 32-bit microcontrollers. It is shown that these cells can be set up to autonomously maintain a schedule and thereby eliminate most of the software overhead involved in time-triggered activations and other time-based mechanisms such as deadline monitoring and execution budgeting. A reference implementation of this design on the Infineon TriCore TC1796 is presented and evaluated along with two commercially available solutions. The comparison shows that this novel concept significantly reduces the overhead in time-triggered operating systems, with up to three-figure speed-ups. Additionally, undesirable effects that could be observed in traditional designs such as situations of priority inversion and unnecessary interrupts are avoided by this new design.

Zusammenfassung

In Standardimplementierungen zeitgesteuerter Echtzeitbetriebssysteme wird der statisch vorkonfigurierte Ablaufplan üblicherweise auf einen einzelnen Hardware-Zeitgeber abgebildet, welcher die CPU zum Zeitpunkt der nächsten geplanten Einlastung unterbricht. In der entsprechenden Behandlungfunktion dieser Unterbrechnung wird der zu aktivierende Task aus der statischen Tabelle herausgesucht, der Zeitgeber auf das darauffolgende Ereignis umprogrammiert, und schließlich der Task eingelastet.

Im Rahmen dieser Arbeit wurde untersucht, inwiefern der damit verbundene erhebliche Rechenaufwand durch den Einsatz von, auf modernen 32-Bit-Mikrocontrollerplatformen in großer Anzahl verfügbaren, Zeitgebereinheiten minimiert werden kann. Es wird gezeigt, wie es durch die geschickte Konfiguration dieser Einheiten möglich ist, die Aufgabe des Abarbeitens eines Zeitplans an das Zeitgebermodul auszulagern, und dadurch die Latenzen zeitgesteuerter Einlastung stark zu reduzieren. Eine Referenzimplementierung für die Plattform Infineon TriCore TC1796 wird vorgestellt, die neben dem zeitgesteuertem sowie gemischt zeit- und ereignisgesteuertem Betrieb weitere zeitbasierte Mechanismen wie Überwachung von Terminen und Zeitbudgets bietet. Eine Evaluation dieser Implementierung im Vergleich mit zwei kommerziell angebotenen Systemen zeigte neben einem vorteilhaftem Laufzeitverhalten durch Vermeidung von Prioritätumkehr und unnötigen Unterbrechungen eine durchweg verbesserte Leistung gegenüber den Vergleichssystemen um bis zu dreistellige Faktoren.

Contents

Chapter 1

Introduction

In time-triggered real-time systems, the central responsibility of the operating system at run-time is to maintain a predefined dispatcher table by activating tasks at specified points in time. Traditionally, this is implemented by multiplexing the various software timers onto a single hardware timer, which is set up to request a scheduling interrupt to the processor each time a new task is planned to be dispatched. The need to decide in software which task is scheduled on a particular timer interrupt and then reconfigure the system timer for the next event makes up for most of the overhead in such a system.

In order to further reduce overheads, this thesis proposes to let arrays of timer cells available on modern microcontroller platforms perform the time-triggered activations of individual tasks without multiplexing a single timer in software. Ideally, all run-time effort of maintaining an execution schedule is thus offloaded from the CPU onto the timer and interrupt periphery.

This approach closely follows the Sloth concept introduced in [1], which endorses the use of existing hardware components for efficient and low-latency scheduling and dispatching.

1.1 The Sloth Concept

The Sloth system is an alternative approach to a task implementation that proposes to model all control flows in an operating system as interrupts, eliminating the traditional distinction between tasks and interrupts. Given an application configuration, Sloth attaches each task to a specific IRQ source on the hardware platform, which is assigned an interrupt priority according to the task configuration. When a task is activated synchronously, the kernel triggers the corresponding IRQ from software. This initiates an arbitration process which the highest priority of all

pending IRQs and compares it to the current CPU priority. If a running task has a higher priority than any pending IRQ its operation will not be disrupted. If, however, a pending interrupt has a priority above the current CPU priority, an interrupt request is sent to the CPU, preempting the running task and dispatching the pending one. This means that all scheduling decisions are performed entirely within the interrupt controller, thus moving a significant part of the kernel responsibilities—and the involved overhead—from the software to the hardware.

The resulting design has been shown to be very light-weight, run with low overheads, and generally outperform traditional software-based kernels by one- to two-figure factors. Despite working at a low level, close to the specific hardware, the simplicity in design allows for a concise implementation that is easily ported to new platforms.

So far, Sloth is restricted to an event-triggered architecture, complying with the OSEK-ECC1 standard for event-triggered real-time systems. The goal of this thesis is to explore the possibilities of expanding into the time-triggered domain while closely adhering to the Sloth design philosophy of minimizing software effort by exploiting peripheral hardware for kernel responsibilities. The resulting system will be called Sloth on Time. Many modern microcontroller platforms are equipped with large arrays of timer cells usually with the intended purpose of generating complex waveforms for motor control appliances. Given that such timer cells regularly act as IRQ sources, too, they appear as viable candidates for autonomously maintaining time schedules in a Sloth system.

1.2 Outline of this Thesis

This thesis is structured as follows. First, some background information on operation system standards with time-triggered architectures is provided in chapter 2, in order to establish the system behavior and features expected by Sloth on Time. Based on these specifications, a set of hardware platform requirements is formulated in chapter 3, leading to a generalized, abstract model of timer cells as they would be available on a suitable platform. Based on these building blocks, chapter 4 presents the design for realizing the mechanisms required for time-triggered execution, followed by an introduction into the timer subsystem of the platform selected for the reference implementation and details on the implementation itself in chapter 5. The resulting system is then evaluated and compared with existing implementations in chapter 6. The thesis concludes with a discussion of related work in chapter 7 followed by a summary and an outlook for possible future work in chapter 8.

Chapter 2

Background

As a reference for time-triggered operating system abstraction, OSEKtime [2] and AUTOSAR OS [3] two widely adopted standards from the automotive industry, have been selected. Both standards are closely related to and share many similarities with the OSEK standard which the existing event-triggered Sloth implementation conforms to. The focus of these standards on automotive applications, however, does not constitute any limitation of the Sloth on Time design regarding the applicability outside of this domain. The choice is mainly based on the simplicity of the interfaces and for practicable comparison with commercially available implementations.

2.1 OSEKtime

The OSEKtime standard for time-triggered systems was published in 2001 for applications in the automotive domain by OSEK, a standards body of a consortium of automotive industry companies. Although it appears as an extension to the more feature-rich, event-triggered OSEK standard from 1995, it describes a system that is independent of OSEK, except for the option to combine both types of systems in a layered manner. This relation and the fact that the existing Sloth design and implementation follows the OSEK standard, makes OSEKtime a suitable reference for incorporating time-triggered elements into Sloth.

OSEKtime offers two types of control flow abstractions, *ISRs* for handling external events and *tasks* that are managed based on time. The activation of the latter is controlled by one or more statically configured dispatcher tables, of which one at a time can be selected to execute at run-time. It is processed cyclically with a fixed run length that is equal for all tables and consists of a set of expiry points at given offsets in time, each initiating the activation of a given task. Tasks are

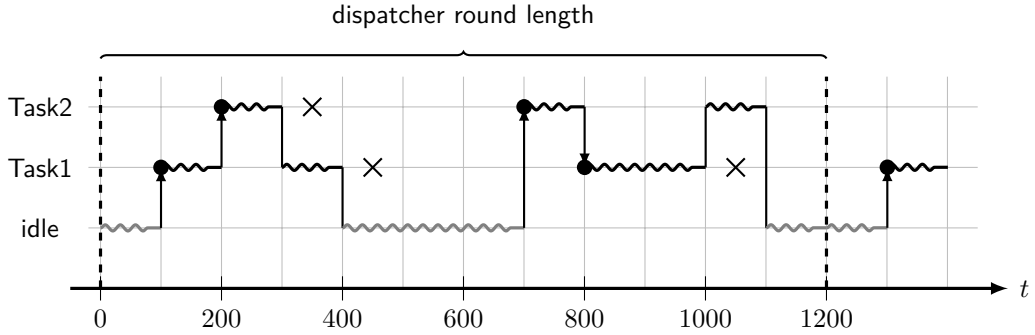


Figure 2.1: The model for time-triggered activation and deadlines in the OSEKtime specification [2]. In this example of a dispatcher table, task activations are depicted by circles, their deadlines by crosses. Later task activations preempt currently running tasks, yielding a stack-based execution pattern.

scheduled in a stack-based manner, such that a task activation invariably preempts any running task, dispatching the new task.

Incorporated in the dispatcher table as well is a mechanism for deadline monitoring. Instead of triggering a task activation at this offset, such a deadline marks a point in time at which its associated task is required to be terminated, otherwise error handling is initiated. Since the same task can be activated multiple times within one dispatcher round, the number of deadlines per task is unlimited as well.

Figure 2.1 shows an exemplary sequence of activations (circles) and deadlines (crosses) in an OSEKtime setup with two tasks. The stack-based execution pattern can be observed in the preemption of `Task1` by a planned activation of `Task2` that coincides with the execution of `Task1` in the first half of the dispatcher round. The opposite situation arises when both tasks are activated in reverse order, as shown in the second half of the dispatcher round.

In distributed environments with multiple nodes communicating over a common bus, it is often necessary for each system to synchronize its dispatcher table against a global time base. OSEKtime describes a very generic mechanism for synchronization, stating that when enabled, the system should regularly check for the occurrence of drift and adjust the execution of the dispatcher table accordingly.

2.2 AUTOSAR OS

In 2005, an international standards committee by the name of AUTOSAR formed to develop a successor to the OSEK/OSEKtime standards. It covers a wide range of aspects in automotive systems, with only the operation system specification

AUTOSAR OS being relevant here. This part of AUTOSAR is strongly based on OSEK/OSEKtime and remains backwards compatible. However, instead of providing a separate, independent standard for time-triggered operation as OSEKtime, AUTOSAR OS seamlessly integrates this domain into its otherwise event-triggered architecture. This leads to an inherently mixed-mode system without any clear distinction between time- and event-triggered control flows. Time-triggered activations to not inevitably preempt any running task like they do in OSEKtime, but instead are regularly included in the scheduling process according to the static priority assigned to each task. This happens within the same priority space used for event-based operation. Furthermore, dispatcher rounds (named *schedule tables* in AUTOSAR OS) are not limited to a single one at a time, running cyclically with a globally defined, static length as in OSEKtime, but can rather be started simultaneously at run-time and have individually configured lengths, optionally being non-cyclic.

Besides time-based activation of tasks, another feature of AUTOSAR OS that is addressed in this thesis is *execution budgeting*, a monitoring mechanism that aims to replace OSEKtime's deadline monitoring. It works by limiting the amount of time a task spends in the *running* state. For this, the system needs to implement facilities that allow to account the budget of the individual tasks at run-time, based on the state each task currently resides in.

Chapter 3

Platform Considerations

With the specification requirements now established, this chapter first outlines the requirements a platform needs to fulfill in order to allow a hardware-based implementation. Then, an abstract model of timer cells is established, providing a generic, hardware-independent basis for approaching the design step in chapter 4.

3.1 Platform Requirements

Two crucial platform requirements to facilitate purely time-triggered task activation in hardware components outside of the central processor can be identified:

- The platform must provide a sufficiently large array of independently operating timer cells, so that all time-based duties of the operating system can be performed by dedicated hardware units, avoiding as much software effort at run-time as possible.
- The provided timer cells need to be able to trigger individually configurable interrupt handlers in order to activate the execution of specific tasks.

When deriving from purely time-triggered paradigms and mixing with the event-triggered components of Sloth, additional requirements need to be fulfilled, which match the demands of a regular Sloth system:

- The platform must offer at least as many interrupt priorities as the amount of priority-controlled tasks required by the application, since each task priority is represented by a dedicated priority level of an interrupt source. If it is not possible for multiple interrupt sources to share the same priority (as it is the case on the Infineon TriCore platform), each time-triggered OSEKtime task

requires a dedicate priority level as well. Two or three additional priority levels are allocated for purely time-triggered OSEKtime operation.

- The platform needs to provide a method for triggering interrupt requests from software as a way to implement synchronous task activations.

The minimum functionality required from each such timer cell consists in the generation of events after a configurable number of clock cycles, with a periodic repetition at a likewise configurable interval. Favorable but not required is the possibility to disable the periodic operation and only trigger a single event at the given timeout. The number of timer cells available has no explicit lower bound. However, in order to go without partially falling back to multiplexing several planned events onto single timer cells and thus requiring regular software intervention at run-time, the number of available cells should naturally correspond to the requirements of the particular application. As it will be shown later, apart from the complexity of the schedule, other factors like enabling execution budgeting or the choice of the method used for deadline monitoring contribute to the eventual demands in terms of array size.

Another non-essential feature that helps to eliminate some more latencies or improve accuracy is the ability to control cell operation for a group of cells at once, this way making it possible to control schedule tables as a whole rather than needing to control its cells sequentially.

3.2 Timer Cell Model

As a basis for developing a system design in accordance to the given standards, this section establishes an abstract model of a timer cell that corresponds to the formulated hardware requirements.

Figure 3.1 includes a block diagram of such an abstract timer cell consisting of an incrementing *counter* field that is continuously compared to a user-defined *compare value*. If both values match and the *request enable* switch is set, the cell requests a hardware interrupt in the attached IRQ source and resets the internal counter to zero. The operation of the entire cell can be halted by disabling the *cell enable* switch, which disconnects the counter register from the cell's clock source. The external unit labeled *control cell* represents the previously mentioned hardware switch allowing the simultaneous control of multiple timer cells at once.

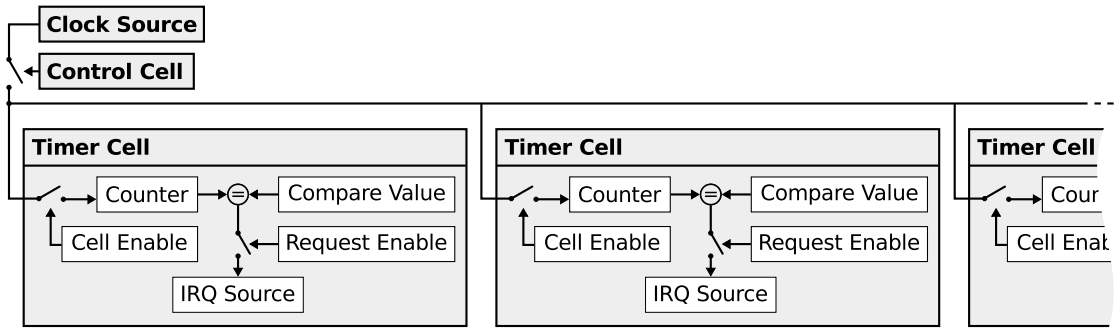


Figure 3.1: The abstract model for available timer components on modern microcontroller platforms, introducing the terminology used in this thesis.

Chapter 4

Design

Taking the specified behaviors of the timer-related components of both OSEKtime and AUTOSAR OS into account and employing the outlined model of timer cells as building blocks, a system design is developed that satisfies the given requirements along with the non-functional concerns of minimized latency and overhead. This chapter first describes a setup for the strictly time-triggered architecture of OSEKtime, then leads over to the modifications necessary to combine this with an event-triggered OSEK system are presented, followed by the adaptation of the AUTOSAR OS specification of integrating time-based activations into an otherwise event-triggered architecture. Afterwards, it will be shown how to realize the additional time-based concepts of schedule table synchronization, deadline monitoring, and execution budgeting.

4.1 General Design Considerations

The main idea of this design is to distribute all timer-related duties among a sufficiently large set of timer cells in a static manner and therefore have the run-time behavior predefined in the initialization phase as much as possible. This aims at avoiding dynamic decisions at run-time, reducing the system overhead and freeing up processor time for the actual user code. The assignment of individual timer cells to the configured objects of an application is part of the configuration, leaving the user in full control of how the available hardware is allocated by the operation system and what is kept available for purposes of the application itself.

4.2 Time-Triggered Operation

On the configuration level, a dispatcher table consists of a list of expiry points, ordered by their time offset relative to the beginning of the table. It has a fixed length and is periodically processed. In a traditional, software-based approach, the schedule of the application is usually represented by a static lookup table in the kernel, which is sequentially traversed from expiry point to expiry point, exactly like it is configured. At run-time, the schedule is maintained by perpetually programming the single system timer for each time a new task activation is planned. This entails the significant overhead of querying a static lookup table for the particular task to be dispatched and the next delay to which the system timer is reprogrammed.

Instead of such a multiplexing mechanism, Sloth on Time employs one timer cell for each expiry point in each dispatcher table, eliminating the need for reconfiguration at run-time for as long as no switching to another table is requested. The proper preparation of the cells consists in the following initialization parameters:

- Set the compare value to the length of the dispatcher table, yielding a cyclic event triggering at this interval.
- Set the initial counter value to the length of the dispatcher table minus the expiry point offset. This way, the length of the expiry point offset elapses before cyclic repetition sets in.
- Set the interrupt number to the IRQ handler corresponding to the task that should be activated, such that a cell event sets the pending bit for this task.

Once the prepared cells are enabled, the correct amount of initial offset elapses in each cell before the first interrupt request, followed by a continuous repetition of interrupts at the interval of the dispatcher table. Without the need for any coordination efforts after initialization, this setup correctly maintains the application's schedule through a set of independently operating timer cells.

Figure 4.1 presents an example setup with a dispatcher table that is 200 ticks long and contains two task activations, `Task1` at an offset of 60 and `Task2` at 170. Both cells employed for this schedule are set up with compare values equaling the dispatcher round length and initial counter values of $200 - 60 = 140$ for the first cell and $200 - 170 = 30$ for the second cell. The two bottom graphs show how the counter value of each cell repeatedly increments over time until it matches the compare value and is reset to zero. This yields two sawtooth patterns of repeated

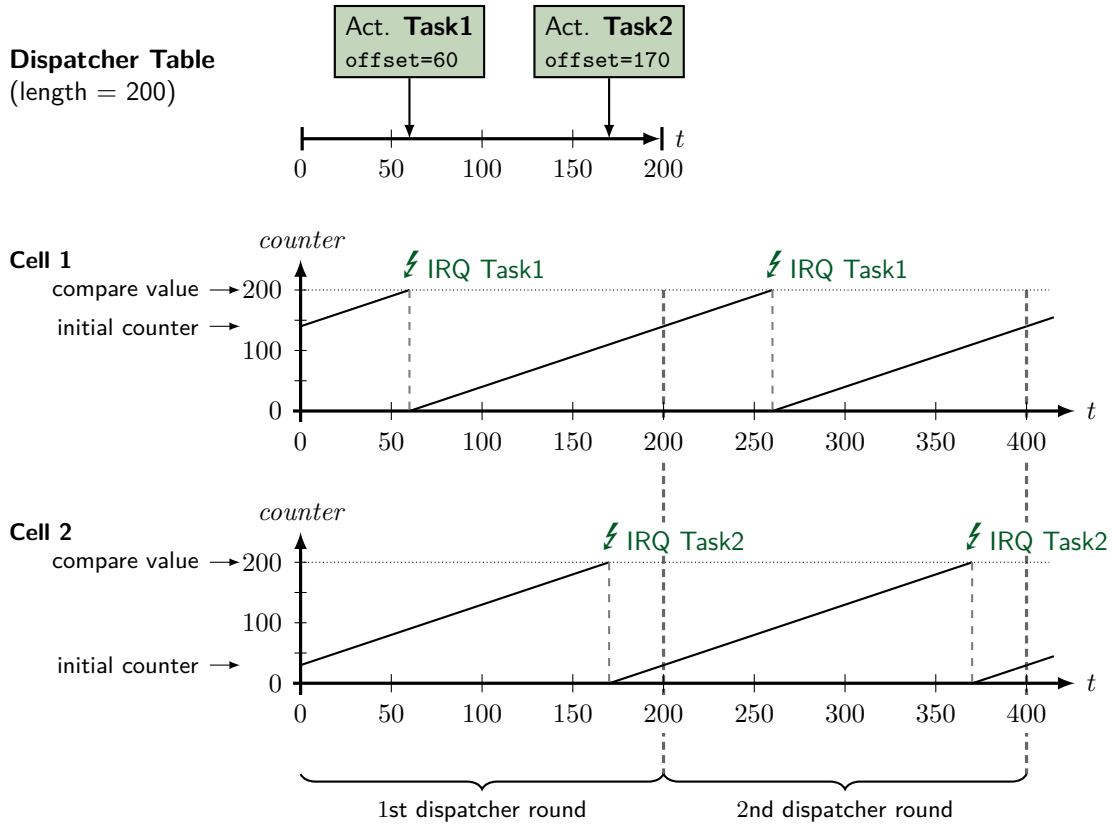


Figure 4.1: Example of the timer cell configurations for a dispatcher table with two time-triggered activations. The initial counter values define the delay of the first interrupt request, which then is followed by repeated requests corresponding to the compare value.

activations, which are aligned on the time axis according to the particular offset of each activation in the dispatcher table.

In order to obtain an execution pattern in which each activation inevitably leads to dispatching the task, the interrupt priority assigned to the handlers of time-triggered activations needs to be set to a level that is guaranteed to be higher than the current execution priority of an already running task. This is done by lowering the CPU priority to a suitable level within the task prologue prior to entering the user code. Due to the run-to-completion semantics of interrupt handlers, this implicitly yields the desired last-in-first-out execution pattern whenever planned activations collide with previously running tasks.

4.3 Mixing Time-Triggered and Event-Triggered Operation

As introduced in chapter 2, both OSEKtime and AUTOSAR OS offer provisions for mixing time-triggered and event-triggered architectures in one application. OSEKtime approaches this in a more separated manner, by allowing to encapsulate an entire event-triggered OSEK system within the idle task of OSEKtime, which gets processor time only if no OSEKtime task is currently running or preempted. Apart from operating on one piece of hardware, there are, however, no points of contact between both systems and no sharing of software resources takes place. AUTOSAR OS, on the other hand, takes a more integrated approach by enhancing its otherwise event-triggered architecture with mechanisms for time-based activation of tasks, without incorporating the principles of a time-triggered architecture, such as strictly stack-based scheduling. This means, regular event-triggered tasks in AUTOSAR OS may additionally be activated by expiry points defined in a schedule table, but nonetheless are subject to priority-controlled scheduling regardless of the source of activation.

In the design of Sloth on Time, the OSEKtime approach of mixed operation is achieved by splitting the interrupt priority space into two sections. The boundary line is defined by an *OSEKtime execution priority* that can be chosen in the application configuration. This priority is the level, all time-triggered tasks of the OSEKtime component in the mixed system will be executed at. In order to ensure the preemptive behavior of time-triggered tasks, the interrupt sources assigned to these tasks are set to an *OSEKtime trigger priority* that is higher than the execution priority. The interrupt handler then immediately lowers the current CPU priority to the execution level before entering the task function. This means, outside of the brief moment between entering the interrupt routine and lowering the CPU priority, all interrupt handlers of time-triggered tasks are guaranteed to have a priority above the current execution priority and therefore always preempt the running task.

Additional attention needs to be paid to critical sections in the kernel. In the regular, purely event-triggered Sloth such sections are secured by temporarily disabling all interrupts. With separated priority spaces for event-triggered and time-triggered tasks, however, disabling interrupts within the event-triggered scope would penetrate the upper half of priority space reserved for time-triggered operation. This issue can be addressed by reserving a priority level between the OSEKtime execution level and the highest level assumed by any event-triggered task. Then instead of disabling interrupts, the current CPU priority can merely be raised to this priority, thereby still allowing time-triggered tasks to preempt the underlying OSEK system.

Figure 4.2 shows an exemplary control flow in a mixed OSEK/OSEKtime setup in Sloth on Time, along with a graph of the CPU priority. Initially, an event-triggered task ET1 is running at its dedicated priority of 1, right below the OSEKtime execution priority of 2. At t_1, the time-triggered task TT1 is activated, triggering an interrupt at the OSEKtime trigger priority of 4. As it can be seen in the bottom graph, the priority level is then lowered to the execution priority of 3 shortly afterwards. This procedure repeats with the activation of another time-triggered task TT2 at t_2. Note how the CPU priority does not change when TT2 terminates at t_3, resuming the preempted context of TT1, which was running at the same priority. On termination of TT1 at t_4, the priority drops below the boundary between the two domains, as the originally running event-triggered task ET1 is resumed.

In contrast to the concept of mixed operation in OSEKtime, AUTOSAR OS is not composed of two clearly separated domains for event-triggered and time-triggered operation, but merely specifies mechanisms for activating event-triggered tasks based on time. In Sloth on Time, this is accomplished by simply omitting the mechanism for lowering the execution priority of time-triggered tasks to a common level. With all tasks sharing a common priority space and keeping the assigned priority after dispatching, no distinction between event-triggered and time-triggered tasks exists as it does in mixed OSEK/OSEKtime operation. The responsibilities of the timer cells used for activations defined by the running schedule table are thereby reduced to setting the pending bit for the appropriate interrupt that is already prepared as part of the regular, event-triggered Sloth system.

4.4 Deadline Monitoring

On the configuration level, OSEKtime's mechanism of deadline monitoring is very similar to the planned task activations. Both are defined by points at a certain offset in the dispatcher table. In conventional, software-based designs they are usually implemented as part of the same static lookup table used for activations and essentially only differ in the code that is eventually executed: instead of entering the task function, a deadline expiry point verifies a task's current state and possibly initiates error handling.

Sloth on Time includes two alternative approaches to deadlines each with its own characteristics with regard to the added overhead and the way they scale with the amount of configured deadlines.

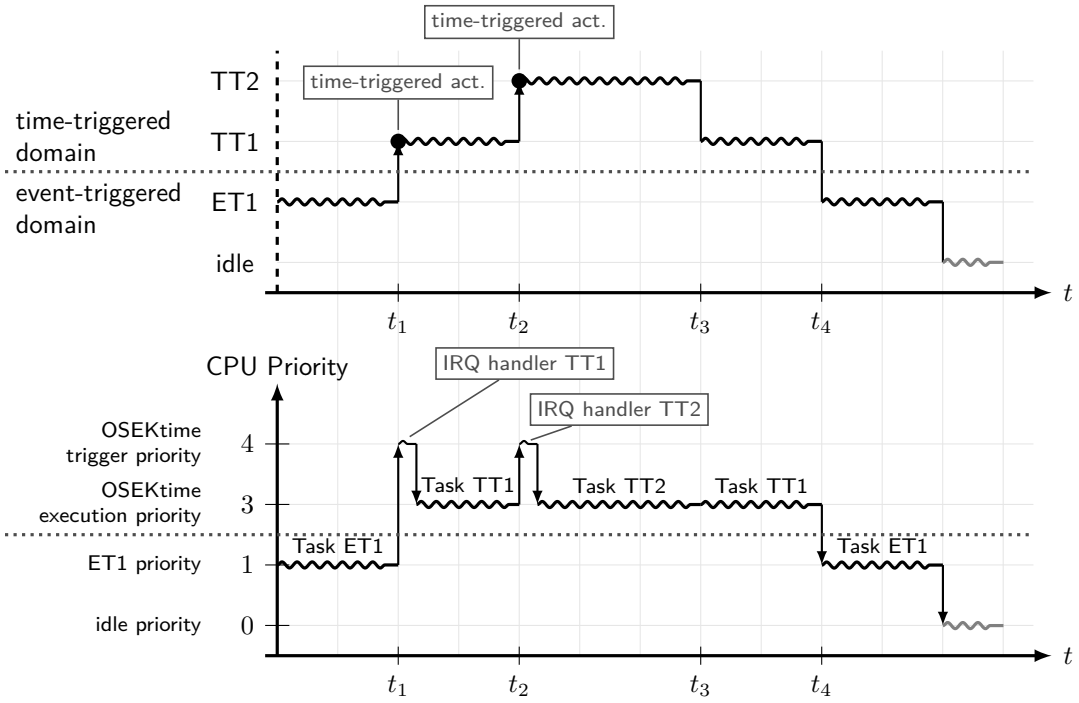


Figure 4.2: Example of the control flow and priority changes in a mixed OSEK/OSEKtime application in Sloth on Time. The event-triggered task ET1 gets preempted by the activation of time-triggered task TT1 at t_1, which in turn gets preempted by another time-triggered task (TT2) at t_2. Both time-triggered activations are performed by a high-priority interrupt request whose handler then lowers the CPU priority to the common execution level of all time-triggered tasks. The priority level does not change when TT2 terminates and TT1 is resumed at t_3. On termination of the only running time-triggered task TT1 at t_4, the event-triggered task ET1 is resumed.

4.4.1 Individual Deadline Cells

For the first method, one timer cell is allocated per deadline. With regard to the initial counter and compare values, the cells are configured identically to regular task activations cells. Their request enable bit, however, remains cleared after initialization and will not be set until the corresponding task is dispatched. For this, the prologue of any task that has deadline monitoring enabled in its configuration will be extended with static code that enables the triggering of interrupts in all cells which represent deadlines for this particular task. When this task successfully terminates in time—that is, without violating its deadline—the equally extended epilogue clears the request enable bits for all deadline cells. This way, when reaching the point in time at which a deadline expires after the monitored task has terminated, it will pass without disrupting the current operation. Consequently, the interrupt

handler attached to a deadline cell does not need to perform any checks against the task's state, but instead directly enters the user-defined error hook routine.

4.4.2 Task-Wise Multiplexing

One apparent disadvantage of the first solution is that the overhead introduced into affected epilogues and prologues grows linearly with the number of deadlines assigned to the same task. This is remedied by the second approach, which employs only a single timer cell for all deadlines of one task and then multiplexes multiple deadlines onto this cell. In a static analysis step during build time, the order of activations and deadlines within one dispatcher round is determined for each task and transformed into a lookup table. The purpose of this table is to provide the run-time environment with the intervals between subsequent deadlines of each task. Using this information, a routine that is inserted into the task epilogue is able to reconfigure the deadline cell such that it represents the next scheduled deadline. This is done by increasing the current compare value of the cell by the interval between the current and next deadlines as it is read from the lookup table.

With this approach, unviolated deadlines no longer imply the uneventful passing of the expiry time due to a cleared request enable bit in this cell. Instead, the deadline cell will never experience a match in counter and compare value for as long as no deadline is violated. Consequently, the clearing and setting of the request enable bit—as it is done in the first method—can be omitted, thereby eliminating any deadline related overhead from the task prologue.

In comparison to the previous method, such task-wise multiplexing introduces a more complex overhead through cell reconfiguration and offset lookups. On the other hand, the overhead does not increase with the numbers of deadlines. Another possible performance advantage stems from the fact that no additional overhead is introduced to the task prologues, but only to their epilogues. While epilogue overhead naturally contributes to the latency of dispatching the task that was previously preempted and therefore is undesirable as well, reducing the latency of task activation generally is preferred over faster task termination.

4.5 Execution Budgeting

Alternatively to deadline monitoring in OSEKtime, AUTOSAR OS specifies its own means for enforcing timing constraints called execution budgeting. Instead of defining fixed points in time at which a task is required to be completed, execution budgeting aims at limiting the cumulative amount of time a task spends in the running state since the most recent activation or release from blocked state. This

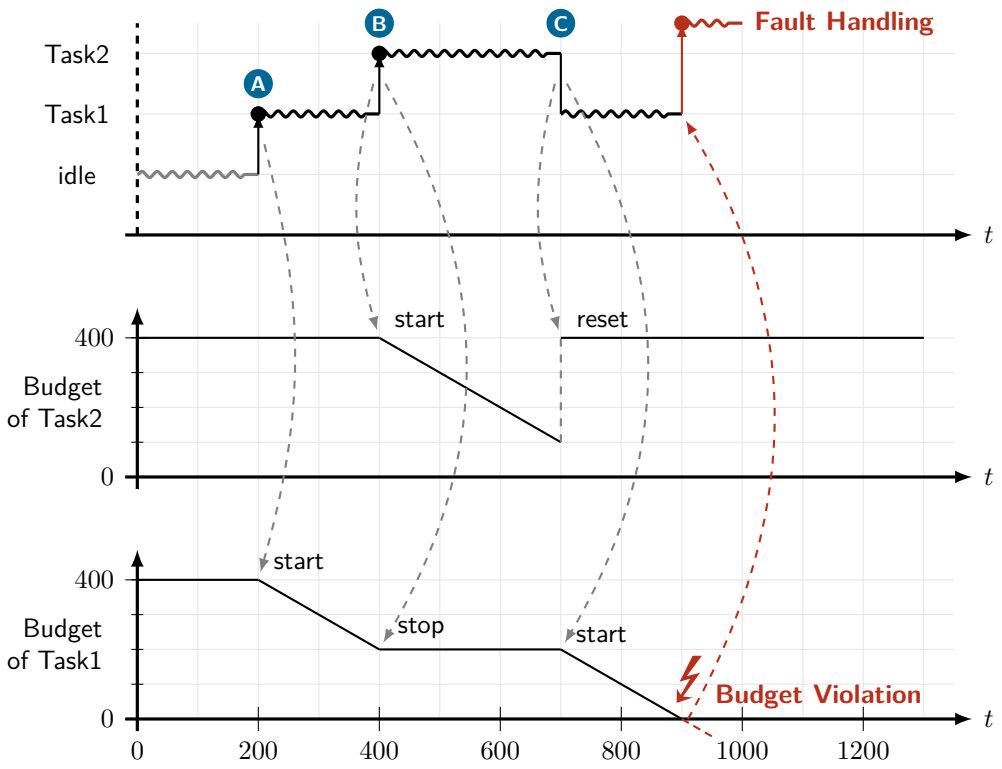


Figure 4.3: Example of an event-triggered application with execution budgeting enabled. The top graph shows the control flow, the dashed lines indicate how task activations and terminations effect the starting, resuming and resetting of the corresponding budget cells.

way, tasks can be monitored regardless of the exact point in time at which they are dispatched.

Here, Sloth on Time employs one timer cell for each task that has a limited budget and attaches the interrupt handler directly to an error handling routine. In contrast to the setups for activation or deadline monitoring, such a cell is not embedded into the time line of any dispatcher round, but operates independently of time-triggered activations. The budget assigned to a task is represented by the difference between the initial values for the counter and compare register. Setting the compare value to the maximum and initializing the counter with the maximum value minus the amount of budget yields a countdown mechanism that automatically triggers error handling on expiry.

In order to properly account for the budget spent by a task, the cell needs to be paused, resumed, and reset at all points of state transitions of this task. For

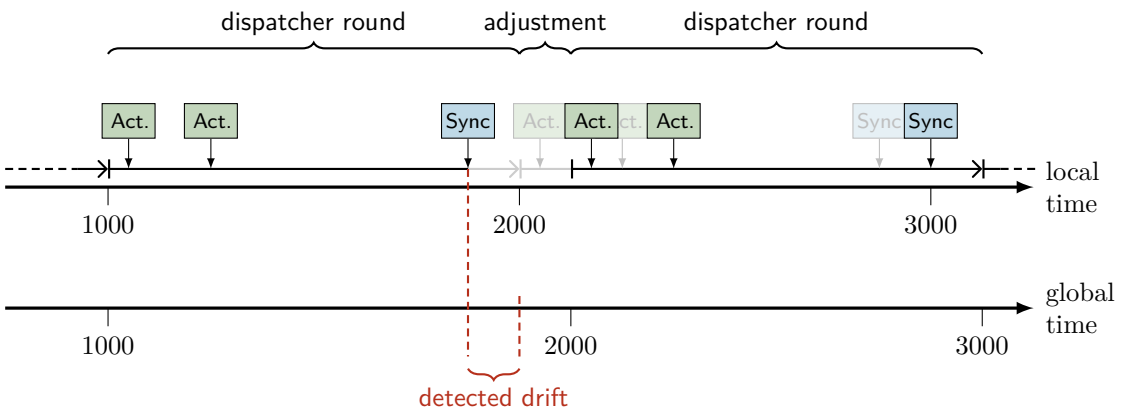


Figure 4.4: Illustration of the OSEKtime synchronization mechanism as it is implemented in SLOTH ON TIME. The synchronization cell at the end of the dispatcher round detects a clock drift, decreases the counter value of all cells allocated by the schedule accordingly, thereby shifting the entiry schedule forward in time.

the task prologues, this means both pausing the budget cell that is monitoring the previously running task, if there is one, and starting the cell associated with the task that is going to be dispatched (see markers A and B in Figure 4.3). On termination (marker C), these two steps are reversed by resetting the cell of the terminated task and resuming the cell of the previously preempted task that is about to be restored. Beyond these extensions to prologues and epilogues, the run-time environment does not need to pay any attention to maintaining counters in software.

4.6 Synchronization

The synchronization mechanism in OSEKtime provides the means for coordinating multiple nodes in a distributed environment, which usually requires the synchronization against a common, global time. This is achieved by frequently—usually at the end of each dispatcher round—determining the current clock drift and then either inserting a suitable delay between the end of one dispatcher round and the beginning of the next, or advance the beginning of the next round. The amount of such a negative drift is naturally constrained by the amount of unallocated time between the last expiry point of one round and the first one of the next round.

Enabling synchronization in Sloth on Time requires the allocation of a dedicated cell in the configuration and a user-provided time offset at which synchronization should be carried out. The choice of this offset affects the possible range of negative drift and can be chosen freely according to the requirements of the application. The cell is prepared almost identically to a regular activation cell except for the attached interrupt handler, which is a special function performing the synchronization. Depending on the current clock drift that needs to be compensated for, this function adjusts all cells that are part of the dispatcher table arrangement (including the synchronization cell itself) in a sequence of read-modify-write instructions adding or subtracting the appropriate value from their respective counter register. In consequence, the next cycle of the dispatcher table as performed by these cells is postponed or pulled forward, thereby adjusting for the clock drift. Although the correction is not applied simultaneously to all involved cells, the procedure accurately preserves the original order of counter values along the time line, since each instruction takes the same amount of time between reading the current value and writing back the modified value. Figure 4.4 provides an example of a dispatcher round with two task activations and synchronization enabled. At the synchronization point, the local time is compared to the global time source, a drift is detected and the three cells assigned to this dispatcher round are adjusted accordingly.

4.7 Summary

The presented design of Sloth on Time unifies the requirements of a time-triggered OSEKtime system and the two different approaches to combining an event-triggered architecture with time-based activations. It also seamlessly integrates with the existing event-triggered Sloth, enabling large parts of the implementation to be shared between the different domains and modes of operation.

It has been shown how the auxiliary time-based mechanisms in OSEKtime and AUTOSAR OS can be realized with individual timer cells as well. For deadline monitoring, an alternative solution that reduces the number of required timer cells has been presented.

Overall, the design achieves the goal of avoiding software efforts at run-time to maintain the configured behavior as much as possible.

Chapter 5

Implementation

This chapter introduces the TriCore TC1796 microcontroller platform, which has been selected for the reference implementation of Sloth on Time. A detailed insight into its general purpose timer array module is provided, while focusing on the components required for the implementation. It then proceeds to the specifics of arranging and configuring the array in compliance with the timer cell abstraction introduced in chapter 3, reflecting the design described in chapter 4.

5.1 TriCore TC1796

Infineon's TriCore architecture is a single-core, 32 bit, RISC design that is specifically oriented towards embedded real-time systems in the automotive domain. The choice for the TC1796, a controller from the TriCore family, is based on two main reasons. First, the existing reference implementation for the Sloth system is already tailored to this platform, eliminating the need for additionally porting the code base to new hardware. Secondly, it fulfills all requirements for Sloth on Time formulated in chapter 3, as will be seen in this section.

Featured as an on-chip peripheral is a sophisticated timer module called *general purpose timer array* (GPTA), which contains several types of timer cell arrays and can trigger a total of 92 different interrupt requests, or *service request nodes*, in TriCore terminology. The following subsection provides a detailed insight into structure and functionality of the GPTA, with an emphasis on the components that are relevant to the implementation of Sloth on Time.

On the top level, the GPTA module consists of three blocks, two virtually identical GPTA kernels, denoted GPTA0 and GPTA1, and one extra kernel called LTCA2, which contains some of the same elements as the GPTAn kernels and is similarly structured but lacks some components and functionality. Figure 5.1 shows

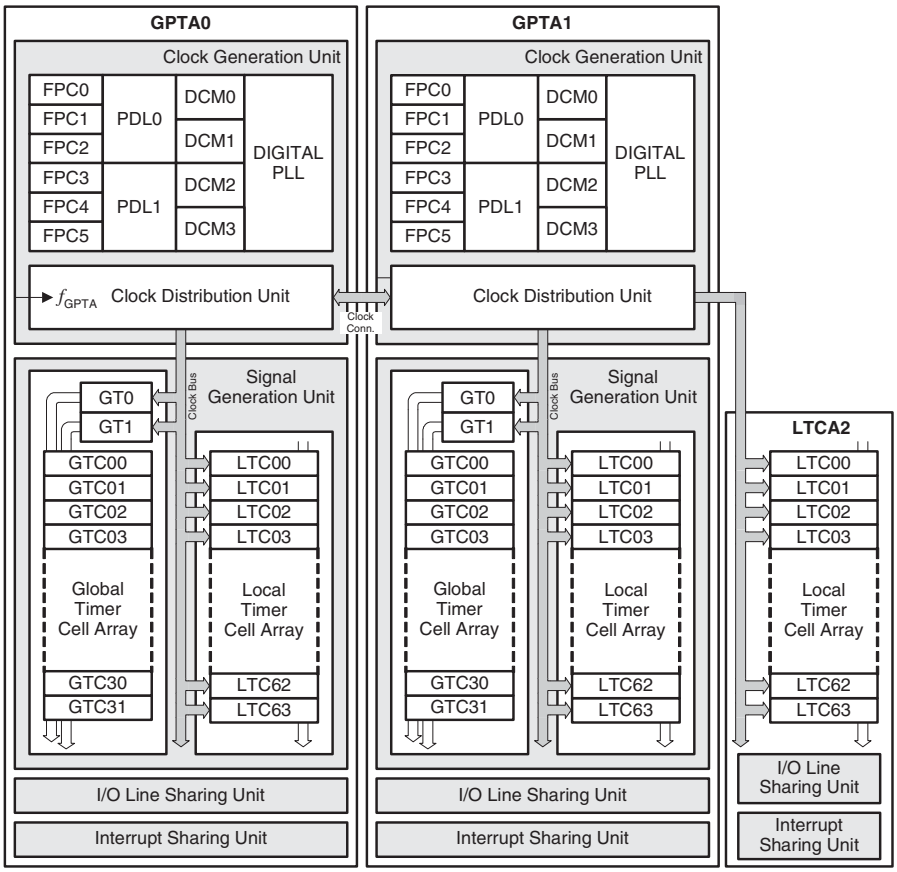


Figure 5.1: Overview of the GPTA module in a TC1796 microcontroller. Taken from the TriCore User's Manual [4].

a block diagram of the complete assembly. Each GPTAn is divided into a *clock generation unit* (CGU), a *signal generation unit* (SGU), an *I/O line sharing unit* and an *interrupt sharing unit*. Following the flow of information, the global GPTA clock signal f_{GPTA} is first fed into the clock generation unit which offers many mechanisms to manipulate the signal into multiple derived clock signals that are eventually distributed among the components of the signal generation unit. The SGU is split up into two arrays of timer cells: one array of 32 *global timer cells* (GTCs) with two *global timers* (GTs), and one array of 64 *local timer cells* (LTCs).

A special role is taken by the I/O line sharing unit. Being interconnected with all other components of the GPTA kernel, it provides the means for flexibly routing the various output lines from the CGU and the individual local and global timer cells to the respective input lines of these units.

Since each of the two GPTA kernels has a total of 111 service request sources—

that is components that are able to trigger an interrupt—but the entire GPTA module only represents 92 distinct service request nodes, the interrupt sharing unit combines up to five sources together into one group and attaches a single service request node to each group. In consequence, the actual request sources are not discernible in software but can merely be attributed to *any* of up to five units. In the case of local timer cells, such a group is consistently composed of four LTCs.

Global Timer Cells

Each GTC is equipped with a 24-bit register and receives two timer values of the same width from the two global timers the array of GTCs is prefaced with. Available output signals are one line to the output multiplexer, two lines to the next GTCs within the array, and a service request line that allows to trigger an interrupt. Analogous to the two output lines going into the next GTC, each cell naturally receives the same two input lines from the previous one, in addition to the regular input signal coming from the input multiplexer. Using the available input data, a global timer cell can operate in two different modes:

Capture Mode: On an input event, save the current value of the selected global timer to the internal register

Compare Mode: Compare a current global timer value with the internal register and generate an output signal or interrupt if the set condition is true.

Local Timer Cells

Unlike GTCs, local timer cells are not connected to a bus of global timer values, but instead maintain an internal timer counter using their internal 16-bit register. Depending on the mode of operation, a timer increase is triggered by an external event, received either from the previous cell in the array or from the input multiplexer (i.e., an actual clock), an external device or a global timer Cell from within the same GPTA module. Figure 5.2 indicates the several input signals and a 16-bit wide register value each cell receives from the previous one. In order to facilitate signaling in both directions along the array, an *event line* going from each cell to the previous one is also available. This interconnected layout makes it possible for a group of cells to cooperate in order to perform tasks that are more complex than what a single cell is able to do. Local timer cells offer several modes of operation, partly resembling the modes of global timer cells:

Capture Mode: On an input event, save the 16-bit value currently read from the preceding cell into the local register.

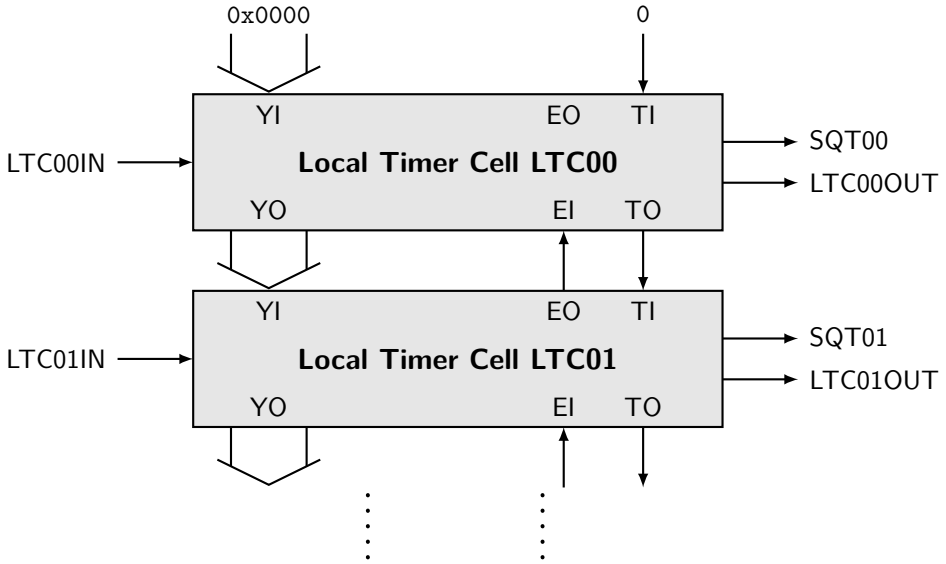


Figure 5.2: Structure of a local timer cell, showing the relations between neighboring cells with the LTC array

Compare Mode: Compare the current value read from the preceding cell with the internal register and generate an output signal or interrupt if both values match

Timer Mode: Increase the internal register value on a certain input signal and optionally generate an output signal whenever an overflow occurs (Free Running Timer Mode). In Reset Timer Mode, it resets its counter to `0xFFFF` if an event is received from the adjacent (next) cell—for example, if that cell is operated in compare mode and detected matching register values.

Each mode allows for a detailed configuration of what will exactly qualify as an input event, what kind of output event will be generated, or if certain behaviors of the particular mode are enabled.

5.2 Timer Cell Implementation

Considering the different modes of operation offered by a local timer cell, it quickly becomes clear that a single cell is not able to implement the behavior required for Sloth on Time timer cells. One LTC can either represent an incrementing counter but then lacks the ability to trigger interrupts at arbitrary intervals. Alternatively,

it can hold a fixed value that might represent an interval length, which is pointless without a current clock counter to compare it with.

The setup developed for Sloth on Time therefore instruments two adjacent local timer cells to act together in order to perform as a fully functional timer cell. In alignment with the groups of four defined by the Interrupt Sharing Unit, the first cell is set to operate in Reset Timer mode. Interrupt request generation when wrapping around is disabled and the I/O multiplexer is set to feed an unmodified clock signal to its input line. The internal register of the first cell represents the counter value in the abstract timer model.

The second cell is put into Compare Mode and holds the static compare value in its internal register. On each timer update happening in the first cell, the internal register is compared to the value read from the first cell. If a match is detected, the second cell triggers an interrupt and prompts the first cell to reset its internal value to `0xFFFF`. After the wrap around in the subsequent cycle, the first cell will then restart counting upwards from zero, repeating the cycle. Taking into account the extra cycles required for resetting and wrapping around, this setup triggers an interrupt precisely every $c + 2$ clock ticks, where c is the value written to the second cell's internal register.

In order to allow the setup to continue counting and resetting along with an entire schedule table but disable the generation of interrupt requests, the *request enable bit* (REN) of the second cell can be cleared. This ensures undisrupted operation of the two cells while prohibiting any further interrupt requests.

For the implementation of the cell enable bit of the abstract timer cell model, an inconspicuous detail of the local timer cells is exploited. Given the fact that the cells are driven by the full speed GPTA module clock, the cells are by construction restricted to process their input signal in level-sensitive mode. With the sampling rate matching the frequency of the input signal, the timer cell ultimately senses a constantly high level on its input line. Consequently, the bit controlling the choice of which level the cell is sensitive to—either active high or active low—directly controls if the timer is incrementing or halting.

Table 5.1 shows the final configuration of both LTCs employed for one functional timer cell. Full documentation of all fields can be found in the TriCore User's Manual [4, p. 24-180 ff.].

As an additional feature that goes beyond the abstract timer cell model, the *one-shot mode* (OSM) field of local timer cells can be used for implementing non-repeating schedule tables. When set, this bit causes the automatic deactivation of the cell at the next internal event. Applying this to all cells of a schedule table that is configured to be non-repeating, yields an arrangement that will halt after performing a single dispatcher round without the need of explicitly stopping the table from software.

Reset Timer Cell		
`MOD = 3`	→	reset timer mode
`OSM = 0`	→	no one-shot mode
`REN = 0`	→	no service request on internal event
`AIL = 1`	→	input signal is active low (cell is initially paused)
`ILM = 1`	→	level-sensitive mode
Compare Cell		
`MOD = 1`	→	compare mode
`OSM = 0`	→	no one-shot mode
`REN = 0`	→	no service request on internal event
`SOL = 1`	→	compare operation enabled on low input level
`SOH = 1`	→	compare operation enabled on high input level
`EOA = 0`	→	enable local events

Table 5.1: LTC configuration values of all relevant bits for two adjacent local timer cells forming a functional timer cell in Sloth on Time

5.3 Coherent Cell Control

Although controlling a timer cell comes as inexpensive as a single write to the memory-mapped control register of the corresponding LTC, it can entail a significant amount of time when a larger group of cells needs to be switched off or on. The system services for starting and stopping schedule tables, for instance, require such management of all cells used for the particular table at once. Furthermore, initiating timer cells in a sequential manner introduces an unintended offset between their counter values, potentially decreasing the accuracy of time-triggered actions.

As a solution for coherently controlling multiple timer cells at once, a setup has been developed that involves the use of global timer cells. Given that the reset timers used in the LTC array do not necessarily require a clock signal on their input line, the I/O sharing unit can as well be configured to connect these cells to the output line of some unit other than a clock. If this signal is then similarly controllable by a simple write into a memory-mapped register, it could be routed to multiple timer cells at once, thus taking the role of a global switch for these cells.

However, the structure of the I/O multiplexing facility imposes a few restrictions on which connections between cells can be established. For instance, it is not possible to route the output signal of an LTC to the input signal of another LTC. In consequence, employing an LTC as a switching cell is ruled out, which leads to

the choice of Global Timer Cells. Additionally, although the output line of one GTC may be routed to the input ports of several LTCs, it cannot be routed to all LTCs but is confined to a certain subset. Besides requiring special attention when configuring an application in order to meet these restrictions, it also limits the number of LTCs involved in such a setup to two groups of 8 consecutive LTCs. Due to the previously shown grouping of four cells into one service request node, this ultimately leaves four functional time cell assemblies to be connected to one GTC.

Another obstacle is the fact that global timer cells do not offer a direct way to set the state of their output line from software, thus requiring a bit of a workaround. According to the specified behavior of a GTC, the output signal is determined by the *output control mode* (OCM) field. This 3-bit vector allows to select if and how the output signal is manipulated whenever an internal cell event occurs. In combination with the *output immediate action* (OIA) bit, which triggers an internal event when written, the GTC output state can eventually be controlled in two steps. First, the OCM is set to either „output line is forced to 0" or „output line is forced to 1". Then, by setting the OIA bit, the selected action is applied, resulting in the output line being set accordingly.

While this setup provides Sloth on Time with the desired functionality of a global switch for multiple timer cells it is restricted by the aforementioned constraints of the I/O sharing unit. Consequently, each schedule table configured to make use of this feature is limited to a maximum number of four expiry points.

5.4 Summary

This chapter has shown how the TriCore TC1796 is a suitable hardware platform for implementing Sloth on Time. It was shown, how the arrays of local timer cells in its general purpose timer array can be instrumented to perform as specified by the timer cell model. Additionally, a setup could be devised that allows the control of multiple cells in groups, switch them on and off in a single memory-write instruction. Applying the Sloth on Time design to the low-level configuration of the timer cells as developed in this chapter, yields a fully functioning Sloth on Time system that meets the specifications of OSEKtime and AUTOSAR OS.

Chapter 6

Evaluation

This chapter covers the evaluation of the Sloth on Time implementation along with a comparison to ProOSEK/Time and tresosECU, two commercially available implementations of OSEKtime and AUTOSAR OS, respectively. First, the setup used for obtaining quantitative measurements of overheads and latencies is introduced in section 6.1. section 6.2 then provides the results of this analysis, followed by qualitative observations made when examining the run-time behavior of each implementation in section 6.3. The chapter concludes with a discussion of these results in section 6.4.

6.1 Evaluation Setup

In order to assess the performance of Sloth on Time, a TriCore TC1796 evaluation board with a Lauterbach PowerTrace unit attached has been used for benchmarking. Besides providing detailed execution traces for observing the preemption patterns in various settings, the device also features built-in functionality for collecting comprehensive statistics on the run-time behavior. Given a benchmarking application and predefined measurement points in the form of code symbols, the scripting interface of the PowerTrace software front end TRACE32 allows to directly obtain the average number of cycles spent within the specified code range. In cases where measurements could be automated this way, a minimum number of 1,000 samples was used in order to get reliable readings. The system was set up to operate both the CPU and the GPTA module at a clock frequency of 50 MHz. Undesirable caching effects were prevented by loading both code and data into the internal RAM, which is not cached.

For the comparison with Sloth on Time, two commercially available solutions supporting the TriCore TC1796 platform have been selected: *ProOSEK/Time* as an

OSEKtime implementation, and *tresosECU* as an AUTOSAR OS implementation. Both systems are a product offered by Elektrobit Automotive and are used by well-known manufacturers such as BMW and VW.

6.2 Quantitative Evaluation

The following section is divided into five parts. The first part addresses the results of the quantitative evaluation of purely time-triggered operation in Sloth on Time compared to ProOSEK/Time. It is followed by the results for mixed operation, with comparisons to both ProOSEK/Time and tresosECU. After that, measurements regarding deadline monitoring, execution budgeting, and the performance of related system services are provided.

6.2.1 Time-Triggered Operation

Looking at purely time-triggered operation, there are two measurable system overheads occurring at run-time. For one, the latency of activating a task is represented by the time that passes between the occurrence of a timer interrupt corresponding to a scheduled task activation and the moment when the first instruction of the user-supplied task function is reached. And second, the overhead of terminating a task and resuming the previously preempted task is obtained by measuring from the last instruction of the task function up to the first task instruction running in the restored context.

A simple application consisting of two tasks, with one task preempting the other, provides both of these transitions. However, special attention needs to be paid to the measurement of the task termination. Since it is not statically predictable which instruction of the preempted task will be the first executed after restoring its context, TRACE32 needs to be instructed to use a range instead of a single code address as the second checkpoint. In this case, the range needs to cover the entire code of the preempted task. This of course implies that the task function must not be left by calling any non-inlined functions at the time it gets preempted.

Figure 6.1 shows the results of this benchmark for Sloth on Time and ProOSEK/Time, given in numbers of cycles. The third row represents the speed-up achieved by Sloth on Time compared to ProOSEK/Time. As anticipated, Sloth on Time only takes the small amount of 14 cycles for both task entry and resuming the preempted task on termination. With 120 cycles for dispatch, ProOSEK/Time is 8.6 times slower, due to the conventional approach requiring significant effort to manage the dispatcher table in software and reconfiguring the system timer. In contrast, task termination in ProOSEK/Time does not require such actions and

	Sloth on Time	ProOSEK/Time	Speed-Up
Time-triggered dispatch	14	120	8.6
Terminate	14	38	2.7

Figure 6.1: Run-time overhead for task dispatch and termination in purely time-triggered OSEKtime systems, comparing Sloth on Time with ProOSEK/Time (in number of clock cycles).

therefore only takes 38 cycles, which is, however, still 2.7 times more than in Sloth on Time.

6.2.2 Mixed Operation

For combining the elements of time-triggered and event-triggered operation in one application, Sloth on Time implements two different approaches. On the one hand, there is the OSEK/OSEKtime solution, which stacks a time-triggered OSEKtime system on top of an event-triggered OSEK system. This is also what is offered by ProOSEK/Time in such way that it allows the integration of the closely related but independently marketed event-triggered ProOSEK system. And on the other hand, there is the approach of the AUTOSAR OS standard, which is essentially an event-triggered system allowing time-triggered task activations. This is implemented in Sloth on Time by omitting the priority space separation and subjecting all tasks to priority-based scheduling. For this variant, the performance of Sloth on Time will be compared to the commercial AUTOSAR OS implementation tresosECU.

OSEKtime

Similar to the benchmark for time-triggered operation, the relevant overheads are the cycles required for time-triggered dispatch and termination with dispatch of the preempted task. In the mixed-system case, however, multiple variants of task transitions arise. Depending on the conformance class, OSEK systems support two types of tasks. *Basic tasks* are restricted to run-to-completion semantics, while *extended tasks* support blocking at run-time in order to wait for an event to be set by another task. For Sloth, the support for extended tasks has been added in the Sleepy Sloth proposal [5]. The Sleepy Sloth design introduces more complex prologues and epilogues for the tasks in an application with support for extended tasks, leading to increased overheads in these cases.

In the benchmarking application for mixed OSEK/OSEKtime operation, this circumstance is reflected in additional measurements, such that each type of preempted

	Preempted State	Sloth	ProOSEK	Speed-Up
Time-triggered task dispatch	idle loop	14	120	8.6
Time-triggered task dispatch	basic task	14	120	8.6
Time-triggered task dispatch	extended task	30	120	4.0
Task termination with dispatch	idle loop	14	38	2.7
Task termination with dispatch	basic task	14	38	2.7
Task termination with dispatch	extended task	30	38	1.3

Table 6.1: Latencies of time-triggered task activation and dispatching in time-triggered OSEKtime systems running on top of an event-triggered OSEK system, comparing Sloth on Time with ProOSEK/Time (in number of clock cycles).

control flow is measured separately. Table 6.1 shows the results and along with an indication for each test case which kind of transition was measured. The results for Sloth on Time in all cases of basic task transitions show the same 14 cycles of overhead as measured before, for both dispatching and terminating time-triggered tasks. The extended task cases show additional 16 cycles for both dispatch and termination in Sloth on Time. ProOSEK/Time exhibits 120 cycles for dispatch and 38 for termination as previously in all three cases, since in traditional designs the task type has no effect on the complexity of these procedures. The speed-ups of Sloth on Time against ProOSEK range between 4.0 and 8.6 in a basic system and between 1.3 and 2.7 in an extended system.

AUTOSAR OS

In AUTOSAR OS, time-based activations can affect both basic and extended tasks. Therefore, the benchmarking application for this scenario covers three types of time-triggered activations: a basic task activation in an otherwise idle system, a basic task preempting another basic task, and an extended task preempting another extended task. Table 6.2 lists the measurement results for these cases in comparison to the same scenario in tresosECU. It is revealed that it has no effect on the overheads in tresosECU whether the involved tasks are of the basic or extended type; it takes 2,400 cycles to activate and dispatch, and 532 cycles to terminate. However, in an otherwise idle system, the overheads reduce to 2,344 for activation with dispatch and 382 for termination. Sloth on Time achieves the regular 14 cycles in all cases except for the extended task cases, in which the more complex prologues and epilogues cause overheads of 77 cycles for the activation, and 88 cycles the termination. Overall, Sloth on Time achieves speed-ups between 31.2 and 171.4 for task activations and 6.0 to 38.0 for task terminations.

	Sloth on Time	tresosECU	Speed-Up
Time-triggered task act. with dispatch idle loop → basic task	14	2,344	167.4
Time-triggered task act. with dispatch basic task → basic task	14	2,400	171.4
Time-triggered task act. with dispatch extended task → extended task	77	2,400	31.2
Task termination with dispatch basic task → idle loop	14	382	27.3
Task termination with dispatch basic task → basic task	14	532	38.0
Task termination with dispatch extended task → extended task	88	532	6.0

Table 6.2: Latencies of time-triggered task activation and dispatching in event-triggered AUTOSAR OS systems, comparing Sloth on Time with tresosECU (in number of clock cycles).

6.2.3 Deadline Monitoring

Enabling deadline monitoring in the application configuration extends the task prologues and epilogues with additional code for enabling and disabling deadline cells, leading to increased overheads. As presented in section 4.4, Sloth on Time offers two different approaches to managing timer cells used for deadline monitoring. The solution that uses individual deadline cells is expected to entail linearly increasing overheads corresponding to the number of deadlines. With task-wise multiplexing of a single cell, on the other hand, the additional overheads should be constant regardless of the number of deadlines.

The measurements listed in Table 6.3 reveal that the anticipated linear increase in the first method settles at around 10 cycles per deadline cell. The resulting overhead for a time-triggered dispatch of a task with four deadlines therefore amounts to 44 cycles; the same applies to its termination.

In contrast, the approach of multiplexing all deadlines of a task onto a single cell entails no additional overhead at all for the task entry, and a fixed increase of 16 cycles on termination. This means, that even in the case of a single deadline per task, the task-wise multiplexing method is the preferable variant in terms of

	Sloth on Time	ProOSEK/Time	Speed-Up
Time-triggered (TT) dispatch	14	120	8.6
Terminate	14	38	2.7
TT dispatch w/ 1 deadline	26	120	4.6
TT dispatch w/ 2 deadlines	34	120	3.5
TT dispatch w/ 3 deadlines	44	120	2.7
Terminate w/ 1 deadline	24	38	1.6
Terminate w/ 2 deadlines	34	38	1.1
Terminate w/ 3 deadlines	44	38	0.9
TT dispatch w/ muxing deadlines	14	120	8.6
Terminate w/ muxing deadlines	30	38	1.3

Table 6.3: Run-time overhead of time-triggered task dispatch and termination with deadline monitoring enabled, comparing Sloth on Time with ProOSEK/Time (in number of clock cycles).

total overhead introduced to both the prologue and the epilogue of the affected task. ProOSEK/Time, in comparison, does not exhibit any effect on overheads by enabling deadlines in the application.

6.2.4 Execution Budgeting

As shown in section 4.5, the execution budgeting mechanism of AUTOSAR OS is approached in Sloth on Time by enhancing the prologues and epilogues with instructions to control dedicated cells, which are each responsible for accounting the spent budget of a particular task. In order to measure the overhead caused by this, a benchmarking application is employed that includes synchronous task activations using the `ActivateTask()` and `ChainTask()` system services, as well as task termination with `TerminateTask()`. Each of these test cases is measured beginning from the service call and ending at the first instruction of the subsequently dispatched task.

The results listed in Table 6.4 reveal that enabling execution budgeting in Sloth on Time increases the run-time overhead of task switches by 26 to 52 cycles. In comparison, tresosECU exhibits additional overhead that is between 13 and 26 times larger than those of Sloth on Time. The total task switch overheads in tresosECU with execution budgeting enabled range between 1209 and 1544 cycles, whereas Sloth on Time requires between 40 and 118 cycles of overhead.

Sloth on Time	budgets disabled	budgets enabled	Δ
ActivateTask() with dispatch	60	91	31
TerminateTask() with dispatch	14	40	26
ChainTask() with dispatch	66	118	52
tresosECU	budgets disabled	budgets enabled	Δ
ActivateTask() with dispatch	768	1476	690
TerminateTask() with dispatch	536	1209	673
ChainTask() with dispatch	856	1544	688

Table 6.4: Run-time overhead of task switches with and without execution budgeting enabled in Sloth on Time compared to tresosECU (in number of clock cycles). The last column provides the difference between the first and second column.

6.2.5 System Services

For the use in applications that require more flexibility at run-time than configuring a set of automatically started schedule tables, AUTOSAR OS specifies two system services for starting and stopping schedule tables at run-time, `StartScheduleTableRel()` and `StopScheduleTable()`. The first one takes an argument controlling the initial delay between calling the system service and the point the schedule table is started. The specification of such a delay or any situation when a previously stopped table is restarted makes it necessary to partially repeat the initialization procedure on the cells allocated by this table. While the configuration bits of the cell remain intact during the execution, initial counter values need to be reset and possibly adjusted according to the initial delay passed to the `StartScheduleTableRel()` call. In the test case created for this measurement, the started and stopped schedule table contains four expiry points and—specifically in the configuration of Sloth on Time—is controlled by a global timer cell for enabling and disabling cell operations. Table 6.5 provides the results yielded in this benchmark, showing a speed-up of 10.2 for `StartScheduleTableRel()` and 37.6 for `StopScheduleTable()` compared to tresosECU. Note, however, that the costs of `StartScheduleTableRel()` in Sloth on Time linearly increase with the size of the involved schedule table if no common control switch for the entire is employed.

	Sloth on Time	tresosECU	Speed-Up
StartScheduleTableRel()	108	1,104	10.2
StopScheduleTable()	20	752	37.6

Table 6.5: Overhead of time-triggered system services in event-triggered AUTOSAR OS systems, comparing Sloth on Time with tresosECU (in number of clock cycles).

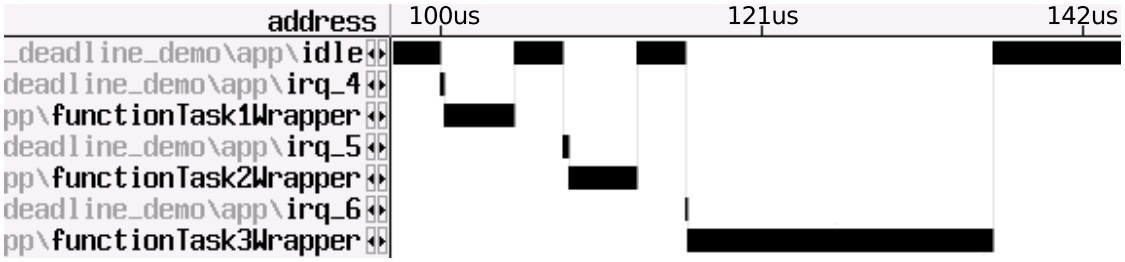
6.3 Qualitative Evaluation

Aside from evaluating the performance of Sloth on Time on the basis of micro benchmarks, a few important qualitative observations could be made as well when examining the run-time behavior of Sloth on Time, tresosECU, and ProOSEK/Time with the help of a hardware tracing unit.

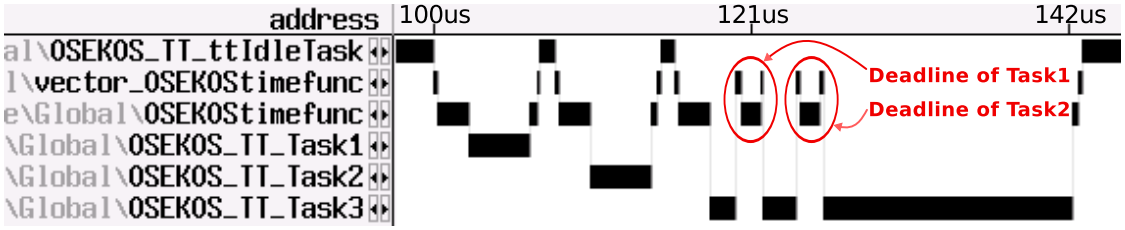
6.3.1 Avoiding Unnecessary IRQs

In section 4.4, the Sloth on Time approach to OSEKtime deadline monitoring was presented, which is based on the idea of canceling the expiry of a given deadline as soon as the corresponding task has terminated and therefore met its deadline. Traditional designs usually perform deadline monitoring by disrupting the current control flow at the point of deadline expiry, checking if the monitored task is running, and resuming the preempted context. In order to verify this assumption and evaluate the impact, a test scenario is laid out in which three tasks are scheduled for execution, one after the other. The first two tasks (`Task1` and `Task2`) have assigned deadlines that expire shortly after the point when `Task3` is planned to be executed. The user-code of `Task3` is made to run long enough to ensure that the expiry of both deadlines coincides with the execution of `Task3`.

Figure 6.2 shows the execution traces obtained from this scenario in both Sloth on Time and ProOSEK/Time. In the trace for Sloth on Time, the execution of `Task1` and `Task2` is seen, followed by an uninterrupted completion of `Task3`. The deadline cells allocated for `Task1` and `Task2` are successfully deactivated upon termination of both tasks and do not trigger an IRQ during the execution of `Task3`. In ProOSEK/Time, however, `Task3` is seen to be interrupted twice in order to perform the deadline checks for `Task1` and Task2. Manual examination of the trace reveals that these interrupts amount to 95 cycles of overhead each and constitute a significant disturbance of the execution of an otherwise uninvolved task.



(a) Sloth on Time

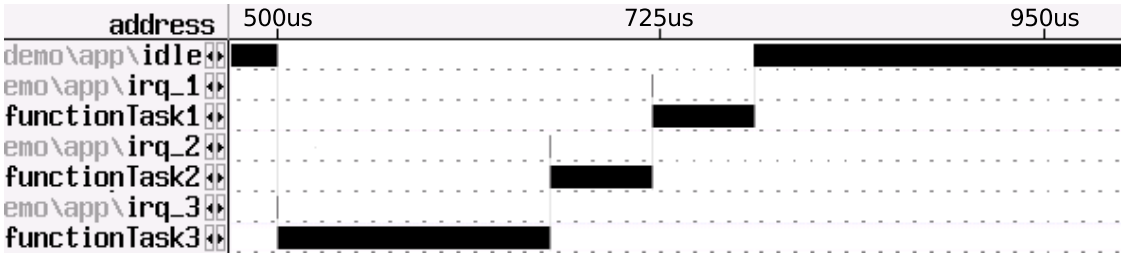


(b) ProOSEK/Time

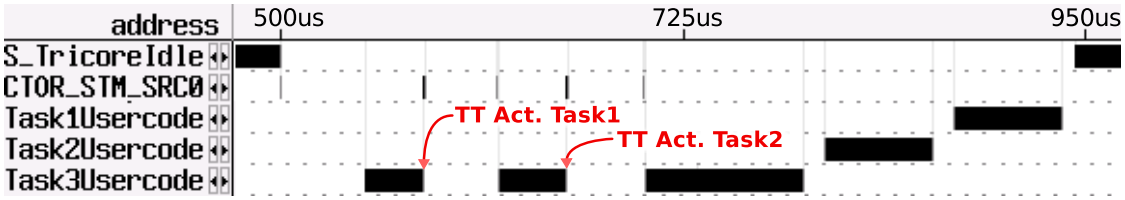
Figure 6.2: Comparison of execution traces of an OSEKtime application with two deadlines in **(a)** Sloth on Time and **(b)** ProOSEK/Time. Non-violated deadlines of `Task1` and `Task2` interrupt the execution of `Task3` in ProOSEK/Time, but not in Sloth on Time.

6.3.2 Avoiding Priority Inversion

When examining the traces of tresosECU, it can be observed that high-priority tasks are routinely preempted by timer interrupts at activation points of tasks with a lower priority. Figure 6.3 gives an example of such a scenario, represented by traces of Sloth on Time and tresosECU. The corresponding schedule table is set up to first activate the high-priority task `Task3` and shortly afterwards—close enough to the activation of `Task3` to ensure overlapping—trigger two tasks of lower priority, `Task2` and `Task1`. The first trace, depicting the run-time behavior of Sloth on Time, shows how this setup correctly pans out to a full completion of `Task3`, then `Task2`, then `Task1`. In tresosECU, however, an interrupt originating from the system timer can be seen at the time of activation of both `Task1` and `Task2`, disrupting `Task3` in its execution for 2,075 cycles each time. This undesirable behavior of keeping a high-priority task from running due to the execution of code on behalf of a low-priority task is known as *rate-monotonic priority inversion* [6]. Sloth on Time manages to prevent this phenomenon by its fundamental design of letting the hardware make the scheduling decisions. Instead of requiring code execution on the CPU to decide if a planned task activation will lead to dispatching



(a) Sloth on Time



(b) tresosECU

Figure 6.3: Execution trace revealing rate-monotonic priority inversion in tresosECU occurring on time-triggered activation of lower-priority tasks. The trace of the same dispatcher table in Sloth on Time shows no interruption of `Task3`.

the task as well, time-triggered activations merely set the corresponding interrupt pending bit and leave it to the IRQ arbitration to decide on interrupting the CPU according to the current execution priority.

6.4 Discussion

The evaluation of Sloth on Time has shown that the approach and design is very beneficial towards the system overhead and the latencies involved in time-triggered operation. It has also shown that mechanisms that go beyond time-triggered scheduling can be efficiently realized in Sloth on Time. For deadline monitoring it could be observed that overheads are introduced at different points in Sloth on Time compared to ProOSEK/Time. While Sloth on Time requires additional efforts in task epilogues but avoids interrupts for non-violated deadlines, deadline monitoring in ProOSEK/Time does not change the performance of task switches but entails interrupts for verifying task states. These interrupts are not needed by the application semantics and were shown to trade significantly more overhead to the interrupted task than deadline monitoring in Sloth on Time does.

The comparison of the two implementation alternatives for deadline monitoring in Sloth on Time has also revealed that employing task-wise multiplexing is

practically always beneficial towards the overall performance. Although in the case of a single deadline, the overhead added to the task epilogue is smaller without task-wise multiplexing, including the prologue overhead in the comparison shows that the task-wise multiplexing method trades less overhead in total, even for a single deadline per task. A remaining drawback of this variant is the additional demand in memory for keeping track of the current position in the offset look-up table, amounting to one 8-bit value per task.

Despite the superior performance compared to traditional implementations, the Sloth on Time design has its limitations. Due to the complexity of a time-triggered application directly relating to the demand in hardware units, the hardware requirements of the application itself compete with the demands of the operating system. In consequence, Sloth on Time may not be applicable in scenarios in which the majority of timer cells is already required for purposes of the application.

Further limitations stem from minor details of the hardware platform but are still propagated up to the configuration level in a way that is not notable in traditional software-based systems. For instance, the 16-bit width of counter registers available in the local timer cells of the TriCore platform imposes a limit on the length of dispatcher rounds and the amount of initial delay when starting schedule tables at run-time. However, when sacrificing timer resolution is acceptable, this could be compensated with the use of a clock prescaler in the GPTA.

Chapter 7

Related Work

There is only little work so far that has focused on improving time-triggered scheduling by hardware assistance. While this might stem from the generally simple and straight-forward implementations of such schedulers [7], there has been ample research on efficient software-based timer abstractions, nonetheless.

Soft timers [8] aim to reduce the overall timer interrupt load by performing timer callbacks on opportunities such as traps and system calls instead of meeting the exact time of expiry. At the end of each system call, page fault handler, etc., the system checks for due timer events and executes the associated handlers, saving the costs of a context switch as it would happen on a separate hardware timer interrupt. The random uncertainty of delaying the timer event past its scheduled expiry is bounded by a periodic hardware interrupt checking for overdue events. As an advantage of this design, the frequency of these periodic interrupts can be lowered, while still maintaining high timer granularity in the average case.

Adaptive timers [9] focus on optimizing the choice of timeout values by observing the timer behavior at run-time and adjusting the timeout values accordingly via a continuous feedback loop. Instead of improving the underlying implementation of timers, this can help reduce the amount of timer events at their source.

The idea of hashed and hierarchical timing wheels [10] addresses the data structures used for maintaining the set of timers. It proposes to perform a variation of bucket sorting in which each bucket covers a larger range of values, such that the farther in the future a timer will expire, the coarser it is sorted. With time moving forward, the timer events are moved to buckets of finer granularity, effectively implementing an incremental sorting mechanism. This way, less effort of sorting all timer events by their expiry is wasted for timers that are prematurely canceled, which is often the case, for example on busy networked servers. This design has been adopted by the timer implementation of the Linux kernel [11].

However, the assumptions of these concepts are that 1) software timers inevitably need to be multiplexed due to the limited availability of hardware timers and that 2) timers and expiry points are a set up dynamically at run-time instead of being statically predefined and that 3) reprogramming hardware timers is costly and needs to be avoided as much as possible [12]. With the availability of timer cells in large arrays on current 32-bit microcontroller platforms, the first assumption does no longer universally apply. While the second assumption holds for general purpose operating systems, it is not valid for time-triggered real-time operation systems, which rely on statically configured schedules with only little flexibility at run-time. Both of these facts allow the Sloth on Time concept to disregard the third assumption by distributing the timer objects of the operation system among dedicated hardware timer cells, eliminating the need for reprogramming timers at run-time.

Chapter 8

Conclusion

In traditional designs for time-triggered real-time systems, the maintenance of a static table for time-based task activations is usually implemented by instrumenting a hardware timer in a fashion that multiplexes the various software timers onto this single timer. In such systems, the effort required to manage the schedule in software and reconfigure the hardware timer after each event constitutes the majority of overhead consumed by the operation system.

This thesis proposed a novel approach to implementing a time-triggered architecture that aims at minimizing software overhead and latencies. It has been shown that, by making use of hardware timer cells available in large arrays on modern microcontrollers, the overhead of managing timers in software at run-time can be eliminated. Instead, all timers and time-based mechanisms on an application are mapped to individual timer cells, which are preconfigured during the initialization phase and autonomously maintain a planned schedule at run-time without the need for software intervention. The devised system not only facilitates the time-based task activation mechanisms specified by the OSEKtime and AUTOSAR OS standards but also incorporates additional timer mechanisms such as deadline monitoring and execution budgeting.

A reference implementation of this design for the TriCore TC1796 platform has been presented and compared in both quantitative and qualitative aspects to two commercial implementations. It has shown to achieve significantly lower overheads as well as exhibit beneficial run-time behavior with regards to unnecessary interrupts and priority inversion.

During the development of the Sloth on Time design and implementation, the TriCore TC1796 has shown to be very well suited as a reference platform for this novel concept. Features such as individually controllable IRQ enable bits per timer cell, and memory-mapped interface for programming the cells in general proved to

be beneficial towards a concise implementation and low overhead when controlling the cells. Despite the close connection between the design and the architecture of the particular hardware platform employed for this thesis, the Sloth on Time concept should be well adoptable to other platforms—for example, the Freescale MPC55xx and MPC56xx embedded PowerPC families—and future platform developments, which possibly differ a lot from the TriCore TC1796 in their design of timer array modules. Adaptations to new platforms potentially require modifications to the assumed timer cell model and subsequently might suggest more abstraction on the application configuration level as well. For instance, the allocation of timer units could be handed to a dynamic mapping procedure that automatically determines a suitable allocation of hardware resources according to the given schedule and configuration of the application.

Bibliography

[1] 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)*, pages 204–213. IEEE Computer Society Press, December 2009.

[2] OSEK/VDX Group. Time-triggered operating system specification 1.0. Technical report, OSEK/VDX Group, July 2001. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`.

[3] AUTOSAR. Specification of operating system (version 4.1.0). Technical report, Automotive Open System Architecture GbR, October 2010.

[4] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TC1796 User's Manual (V2.0)*, July 2007.

[5] 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)*, pages 67–77. IEEE Computer Society Press, December 2011.

[6] 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)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[7] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Englewood Cliffs, NJ, USA, 2000.

[8] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, August 2000.

[9] 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)*, pages 205–218, New York, NY, USA, March 2008. ACM Press.

[10] G. Varghese and T. 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)*, pages 25–38, New York, NY, USA, 1987. ACM Press.

[11] Ingo Molnar. kernel/timer.c design. `http://lkml.org/lkml/2005/10/19/46`, 2005.

[12] Antônio Augusto Fröhlich, Giovani Gracioli, and João Felipe Santos. Periodic timers revisited: The real-time embedded system perspective. *Computers & Electrical Engineering*, 37(3):365–375, 2011.

List of Figures