Doctoral theses at NTNU, 2012:78

Kristoffer Nyborg Gregertsen

Execution time control

A hardware accelerated Ada implementation with novel support for interrupt handling

Doctoral Thesis

Kristoffer Nyborg Gregertsen

ISBN 978-82-471-3429-0 (printed version)
ISBN 978-82-471-3430-6 (electronic version)
ISSN 1503-8181

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology
Thesis for the degree of Philosophiae Doctor
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Engineering Cybernetics

Doctoral theses at NTNU, 2012:78

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU

Kristoffer Nyborg Gregertsen

Execution time control

A hardware accelerated Ada implementation with novel support for interrupt handling

Thesis for the degree of Philosophiae Doctor

Trondheim, April 2012

Norwegian University of Science and Technology
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Engineering Cybernetics

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Engineering Cybernetics

ISBN 978-82-471-3429-0 (printed version)
ISBN 978-82-471-3430-6 (electronic version)
ISSN 1503-8181

ITK Report 2012-2-W

Doctoral theses at NTNU, 2012:78

Printed by Skipnes Kommunikasjon as

Til Christine og Edvard.

Summary

Execution time control is a technique that allows execution time budgets to be set and overruns to be handled dynamically to prevent deadline misses. This removes the need for the worst-case execution time (WCET) of tasks to be found by offline timing analysis – a problem that can be very hard to solve for modern computer architectures. Execution time control can also increase the processor utilization, as the WCET will often be much higher than the average execution time.

This thesis describes how the GNU Ada Compiler and a bare-board Ravenscar run-time environment were ported to the Atmel AVR®32 UC3 microcontroller series making the Ada programming language available on this architecture for the first time, and an implementation of Ada execution time control for this system that supports full execution time control for interrupt handling. Usage patterns for this brand new feature are demonstrated in Ada by extending the object-oriented real-time framework with execution time servers for interrupt handling, allowing the system to be protected against unexpected bursts of interrupts that could otherwise result in deadline misses. Separate execution time measurement for interrupt handling also improves the accuracy of measurement for tasks. As a direct result of the work presented in this thesis separate execution time measurement for interrupts will be included in the forthcoming ISO-standard for Ada 2012.

While the implementation of execution time control is for the Ada programming language and the UC3 microcontroller series, the design and implementation should be portable to other architectures, and the principles of execution time control for interrupt handling applicable to other programming languages.

Low run-time overhead is important for execution time control to be useful for real-time systems. Therefore a hardware Time Management Unit (TMU) was designed to reduce the overhead of execution time control. This design has been implemented for the UC3 and performance tests with the developed run-time environment shows that it gives a significant reduction of overhead. The memory-mapped design of the TMU also allows it to be implemented on other architectures.

Contents

Preface

The research work leading to this thesis was funded by a PhD grant given by the Faculty of Information Technology, Mathematics and Electrical Engineering at the Norwegian University of Technology and Science (NTNU). Getting this grant allowed me the great privilege of choosing a research topic after my own interest. This allowed me to continue my research on execution time control for the Ada programming language that I started with my master thesis. I wish to express my gratitude to the Norwegian taxpayers who in the end have financed my work, and believe my work is a contribution to more reliable embedded systems.

My research work has also been a journey through Norway. I started working on this thesis after finishing my master degree in 2008, when my spouse and I moved to the beautiful small town of Ålesund for a year. Luckily, Ålesund University College provided me with an office, and a stimulating work environment during this critical first phase of my research. After the year in Ålesund, we moved to the little village Ørsta. We lived there between the fjords and mountains for one and a half year. While it sometimes was hard to work in the solitude of my home office, living with the amazing nature literally right outside the door made up for it.

In 2011 we moved back to Trondheim and I could finally be with colleagues again. Shortly after we got our first child. Becoming a father also meant little sleep at night and lots of work, but I would not change it for anything in the world. In the fall the final pieces of the research work were put into place, and we also moved into our first house. This thesis was completed while I stayed home with my son which was quite challenging, having to write in stole moments while he slept at daytime, in evenings, and during weekends. Luckily, the thesis was mostly finished before this, and I could enjoy the time with my son without worrying to much about it.

Acknowledgments

This thesis would never have been written without the support of my advisor, Amund Skavhaug. He recruited me already in the third year of my master study, and guided me in selecting relevant courses within real-time systems, computer architecture and electronics. It was also Amund who encouraged and helped me to apply for the PhD grant, and helped me select the research topic. During my PhD work he always supported me with my research, but also with more personal matters. He has reminded me that there are other things to life than just work. Amund often talks about his past as a badminton coach, and how he has adapted the coach attitude in guiding his students. One can really tell that this is true by the quality of his work. You are the best advisor I could ever hope for, Amund.

My gratitude also goes to the companies Atmel Norway and AdaCore, who have supported me in my research effort. Atmel Norway has provided hardware and open-source development tools, and allowed the TMU to be implemented and tested with the AVR32 UC3 microcontroller. Special thanks to Ronny Pedersen for supporting GCC for AVR32, master student Stian Søvik at NTNU for implementing the TMU, and Frode Sundal and Martin Olsson for helping me with the testing. Thanks to AdaCore for providing the open-source version of GNAT that has been used throughout this research. Special thanks to José F. Ruiz for helping porting the Ravenscar run-time to the AVR32.

I wish to thank Ålesund University College for providing me with an office for the year I stayed, and all the nice people there for being so welcoming and inspiring. You always made me feel like one of you.

Another bunch of kind people are those of the Ada real-time community, many of whom I met at IRTAW-14 in Portovenere. I felt welcomed and appreciated among you, and hope to meet you all again in the years that come. Special thanks to Alan Burns for encouraging me to participate at the workshop, and Juan de la Puente and Juan Zamorano for taking me to see “Cinque Terre”.

I would like to thank my family who has always supported and believed in me. My mother and father have nourished my curiosity through my childhood, taking me to the museum of natural history in my home town Stavanger about every week. After walking that much among stuffed animals I probably should have become a zoologist, and I would therefore like to thank my uncle Knut for introducing me to computers. It may have been the turbo switch on your 486 that triggered the process leading to a PhD in real-time systems.

Many thanks to my good friends Henrik and Robin for proof-reading the thesis.

Finally, I would like to thank the love of my life Christine, without whom I could not have finished the PhD with my sanity intact. You have always encouraged me in my research work, while making sure that this work did not expand to fill my whole life. You have also provided good reality orientation when needed. We have also been blessed with a beautiful son Edvard, who has given my life a whole new perspective and provided yet another reason to get the PhD done.

Chapter 1

Introduction

1.1 Background and motivation

Embedded systems are everywhere in today's society – to such an extent that we take their functionality for granted. They are present in equipment such as mobile phones, washing machines and televisions that we use every day, and in automobiles, airplanes and rail-road systems upon which our lives depend when we use them. A sign of this omnipresence is that the sale of processors for embedded systems far outweigh that of processors used in personal computers and servers. Common for embedded systems is that they are an integral part of the larger system that depend on its functionality. Therefore, the malfunction of an embedded system may cause great economic losses, or in the worst case, even result in material damage and loss of life.

The term *dependability* describes the systems' ability to provide its functionality according to specification. The system experiences a *failure* when this functionality cannot be provided. A failure is caused by an *error* that is a manifestation of an underlying *fault* in the system. Faults may be physical like component wear-out; transient, like electromagnetic interference; intermittent that come and go for instance due to components overheating; or logical human-made faults in hardware, software or the interaction between these. To achieve the desired level of dependability one can apply techniques to prevent faults from being introduced in the design. This is known as *fault prevention*. However, as it is hubris to declare any system fault-free, it is equally important to make sure that errors does not cause failure. This is known as *fault tolerance*.

1.1.1 Real-time systems

As embedded systems often interact with the physical world, they have temporal requirements to the produced output in addition to the value of the output itself. Thus, if the output is not produced on time it is an error in the timing domain. Such systems are commonly known as *real-time* systems.

A *task* is an entity that performs some work in the system, and is *released* either periodically or by some sporadic event. Each release is called an *instance* of the task. Real-time tasks also have an associated *deadline* relative to the release time for finishing their work and producing the output. Deadlines are commonly classified as *hard* if a deadline miss is an error which may result in system failure, *firm* if a missed deadline means that the result has no value but does not cause failure, and *soft* if the value of the result gradually drops after the deadline. In this work deadlines are considered hard unless explicitly stated otherwise.

For practical and economical reasons a set of tasks is usually executed on the same computer, sharing the processor cores and other resources such as data and hardware units among them. The resources, and in particular the processor execution time, are shared among the tasks according to some *scheduling policy*. A scheduling policy is *static* if the task priorities are always the same, and *dynamic* if it changes at run-time. *Scheduling analysis* is applied to make sure that system is *schedulable* which means that all tasks will be able to meet their deadlines.

1.1.2 Worst-case execution time (WCET)

Scheduling analysis requires the worst-case execution time (WCET) of tasks to be known. The WCET is computed by applying *timing analysis* on the task, either by static analysis of the source code and the compiled executable using an abstract model of the computer architecture, or by measuring the execution time of the task or parts of the task when executed on the targeted hardware [62]. Finding the WCET may be hard for all but the most trivial tasks as a large space of input data and initial conditions need to be considered. Furthermore, timing analysis is made magnitudes harder by performance enhancing techniques such as multi-level cache, deep pipelines with shared execution units, and more. These techniques introduce *timing anomalies* that can be counter-intuitive and hard to predict [62]. Timing analysis is even harder for multi-core computer architectures as the execution on the different cores will be affected by how the others use the coherent cache hierarchy and other shared resources.

The *precision* of the timing analysis is how close the *computed* WCET is to the

actual WCET, while the *safety* tells if the timing analysis provides a guaranteed bound for the actual WCET or just a prediction [62]. For hard real-time systems safety is paramount as deadline misses could be the result if the WCET used for scheduling analysis is less than the actual WCET. The overestimation of state-of-the-art timing analysis tools is reported to be in the range of 30-50% [62]. Furthermore, even if a computed WCET is very precise, the actual WCET can be magnitudes higher than the average execution time, since it includes the unlikely event of several performance enhancing techniques failing at the same time. Therefore, using the WCET as the tasks execution time *budget* leads to waste of computational resources as the tasks usually will require much less execution time than budgeted for in the scheduling analysis. For some systems there may be soft tasks that can utilize the additional available execution time, but this may not always be the case.

A survey published as recently as 2008 describes several state-of-the-art methods and tools for timing analysis, both prototypes and commercial [62]. Out of the many tools listed, all but one assume uninterrupted task execution due to the effects interruption has on the state of the cache, and can therefore not be used with preemptive scheduling. Furthermore, none of the described tools work with multi-core processors with a shared cache. The tools also require detailed descriptions of the hardware used, and are therefore limited to a handful of computer architectures used in embedded systems. For high-end architectures using performance enhancing techniques the timing analysis of a task can take as much as a day [62]. While the survey reports that the problem of finding safe bounds to WCET is solved, and that the described tools has found successful use, this use seems to be limited to high-integrity projects with large budgets and long development time, such as the aviation, automotive, defense and space industry.

Timing analysis has been subject of much research effort since the survey of 2008, and recent research does try to address the multi-core WCET problem [16, 35].

1.1.3 Execution time control

Execution time control allows dynamic control of the tasks execution time instead of solely relying on static guarantees, and thus provides fault tolerance in the timing domain instead of just fault prevention. This is done by setting an execution time budget for the tasks at release time and handling budget *overruns* according to some application-dependent *policy* to prevent deadlines being lost [60]. Note that execution time control requires some form of timing analysis, as knowledge of the execution time properties is needed to set reasonable task budgets. However, as safety is provided by execution time control, the requirements for the timing ana-

lysis are less strict – guaranteed bounds on the WCET are no longer needed. This means that simpler measurement based techniques can be used to find a reasonable budget for the tasks.

Execution time control is also a prerequisite for execution time servers such as the deferrable and sporadic server that allows for soft sporadic tasks to have short average response time whilst guaranteeing for the deadlines of hard periodic tasks [14]. Furthermore, execution time control facilitates tasks executing algorithms where there is an increasing reward with increased service (IRIS) [36]. In this case, the algorithm is stopped when it has converged or its execution time budget is exhausted. If no acceptable result was computed in time a simpler algorithm may be executed.

The *mechanism* for execution time control is provided by the run-time environment and measures the time a task has been executing on the system by providing a high accuracy execution time clock that can either be explicit or implicitly defined in the system. Execution time monitoring allows applications to provide *handlers* for a clock that are called when the execution time of the clock reaches a specified timeout value. The mechanism for this is referred to as an execution time *timer*, and is a type of *alarm*. The run-time overhead incurred by the mechanisms should be as low as possible for execution time control to be acceptable for use in real-time systems.

1.1.4 Interrupts

Interrupts cause the normal execution of tasks to be paused and a *handler* to be executed, either as a result of an asynchronous hardware interrupt line being asserted or a synchronous software interrupt being triggered. An interrupt is said to *occur*, and an occurrence is *pending* in the time between its *generation* and its *delivery* to the system in the form of the appropriate handler being called. It depends on the hardware whether a generated interrupt occurrence is lost if another of the same type is already pending.

This work deals with *hardware* interrupts unless explicitly stated otherwise. These interrupts may be generated by components of the computer system such as peripheral units, or by external sources. Often the computer architecture has an interrupt controller that multiplexes and groups interrupt lines, and triggers the interrupt handling on the processor. There may also be several interrupt *levels*, where interrupts may be interrupted by others of a higher level. With the exception of non-maskable interrupts, the delivery of interrupts may be blocked by the use of *masks*. Whether a blocked interrupt remains pending or is lost depends on the

architecture and hardware.

The execution time of interrupt handlers has usually been charged to the interrupted task. This causes inaccuracy in the execution time measurement of tasks, which again means that the budgets of all tasks have to be extended to allow for the additional execution time of interrupt handlers as it cannot be known in advance which tasks will be interrupted. Furthermore, as interrupt handling have higher priority than normal task execution, deadlines may be lost in the case of an unexpected high rate of interrupts, either due to a design or analysis fault, or an error in hardware generating more interrupts than the system can handle. While it is possible to count the number of occurrences and deduce the execution time from these, this is an inaccurate method and relies on the WCET of the interrupt handlers being known.

The lack of proper protection against unexpectedly high rates of interrupt occurrences motivates the development of execution time control for interrupt handling similar to that for tasks. The overhead to interrupt handling has to be very low for this new feature to be usable in real-time systems, and specialized hardware may therefore also be needed.

1.2 Research goals and methods

The primary goals of this research are:

1. To establish a research platform by porting the GNU Ada Compiler (GNAT) and a bare-board run-time environment to the Atmel AVR32 UC3 microcontroller series and implement execution time control for this system.

2. To design and implement execution time control for interrupt handling and demonstrate usage patterns for this new feature in the Ada programming language.

3. To design and implement a dedicated hardware Time Management Unit (TMU) for reducing the overhead of execution time control.

The Ada programming language was chosen for this research work as it is renowned within high-integrity systems, has built-in real-time tasking support, and has supported execution time control for tasks since Ada 2005 [32]. Also, there is a fairly small but active community researching on Ada for real-time systems, with the International Real-Time Ada Workshop (IRTAW) being an important meeting place for proposing and discussing new real-time features for the language.

GNAT is a front-end for the GNU Compiler Collection (GCC) maintained by AdaCore. Both GCC and GNAT are open-source software available under the GNU Public License (GPL) [54] and the research can therefore be shared freely, benefiting the whole community. The GNAT bare-board Ravenscar run-time environment [47] was used as real-time kernel for the research. Due to the limited tasking support of the Ravenscar profile [10, 11], the run-time environment is small in code size and is therefore easy to understand and make changes to. The run-time environment is also of high quality and forms a solid base for the research.

Due to the close relation between NTNU and Atmel Norway, it was decided to use their new AVR32 architecture and the UC3 microcontroller series [3, 4, 57] as a hardware platform for the research. This allowed for the TMU to be implemented and tested with Atmel's proprietary synthesizable code for the UC3. Also, there is an open-source GCC back-end available for AVR32 so that GNAT can easily be ported to the architecture.

1.3 Related work

Execution time control is supported in different ways by many systems. For decades mainframe computer systems have allowed setting execution time budgets for jobs and users in order to protect and share the valuable processing time. Often general purpose operating systems have a periodic scheduling tick in the frequency range 10 to 1000 Hz, and will find a statistical *approximation* of the execution time by counting which process is running when this tick handler is called. However, the coarse-grained precision and uncertainty of this execution time measurement method makes it unusable for real-time systems.

Real-time POSIX has supported execution time control since POSIX.1b standardized in 1993. It defines execution time clocks for processes and threads, and timers to signal overruns for these clocks [20,55]. Using these POSIX features Harbour et al. at the University of Cantabria implemented execution time control for Ada 95 and demonstrated usage patterns for real-time applications [21]. The same research group also implemented and demonstrated these features on the embedded MaRTE OS [45]. The execution time control features were proposed added to the Ada language standard and discussed at IRTAW-10 [44]. The proposal was later refined [41] and was forwarded by IRTAW-12 to the Ada Rapporteur Group (ARG) for the process of ISO-standardization [1].

Execution time control was standardized together with other new real-time features in Ada 2005 [32]. However, the standard did not state which execution time

budget, if any, that is to be charged the execution time of interrupt handlers. All implementations known to the candidate when this research work started, charged the running task [17,21,22,45,56]. This causes inaccuracy to execution time measurement and was pointed out as an issue when the new Ada 2005 real-time features were evaluated [59]. At IRTAW-14 where the candidate proposed adding full execution time control for interrupt handling [24], the developers of MaRTE OS also proposed adding a execution time clock for all interrupt handling combined, primarily to improve the accuracy of execution time measurement for tasks [46]. These two independent proposals were forwarded to ARG for the process of ISO-standardization [40] and are to be included in the forthcoming Ada 2012 standard [33].

Execution time control was also implemented for the Open Ravenscar Kernel (ORK) by de la Puente and Zamarano at the Polytechnical University of Madrid prior to standardization of the feature in Ada 2005, and execution time control policies within the limitations of the Ravenscar profile were demonstrated [17]. As the Ravenscar prohibits asynchronous task control and changes of priorities, most of the policies possible with the full Ada 95 tasking model [21] could not be used, leaving only overrun detection and system reconfiguration as options. However, it was concluded that execution time control indeed could be useful with the Ravenscar profile [17]. Yet, when execution time control was standardized with Ada 2005, execution time timers were explicitly prohibited with the profile [32] primarily because of the static nature of Ravenscar tasking and the lack of mechanisms to handle overruns. Still, ORK and the GNAT bare-board kernel based on it [47] have continued to support execution time control [56].

1.3.1 Other languages and systems

The Real-Time Specification for Java (RTSJ)[1] supports execution time control by an integrated approach referred to as *cost monitoring* [50]. In essence the RTSJ allows budgets, or the *cost*, to be set for periodic threads. This cost is the same for each release. In the case of a cost overrun the offending thread will only be allowed to continue executing if this will not cause lower priority threads to miss their deadlines, otherwise it will be immediately blocked until its next release. The cost monitoring scheme is intended to be independent of scheduling policy [50].

The QNX® Neutrino® RTOS[2] supports execution time control by allowing reservation of CPU time for *partitions* consisting of processes and tasks. The OS uses

[1] Web: `http://www.rtsj.org`

[2] Web: `http://www.qnx.com/products/neutrino-rtos`

an approach that is adaptive in that CPU time not used by one partition may be utilized by others. The limits on CPU time are only enforced when the system is overloaded in order to guarantee that the reserved CPU resources are available for specified processes. Partitions are configured, not programmed, and are therefore flexible in use, no recompilation is needed in order to change the allocation of CPU resources.

1.4 Thesis organization

The reminder of this thesis is organized as follows:

Chapter 2: Theoretical background of the thesis, giving a brief introduction to real-time scheduling systems and execution time control; the Ada programming language and GNAT; and the Atmel AVR32 architecture.

Chapter 3: Description of the main contributions of this PhD in the form of published and submitted material.

Chapter 4: Conclusions on the main contributions of this PhD and a brief discussion of future work topics.

Appendix A: The published and submitted material forming the basis of this thesis in chronological order.

Appendix B: The presentations of interrupt execution time control by the candidate at IRTAW-14 that formed the basis for ISO standardization of this new feature.

Appendix C: Description of how to obtain the sources for GNATforAVR32 and set up the development environment.

Appendix D: Additional code listings.

Chapter 2

Background theory

2.1 Real-time systems and execution time control

2.1.1 Scheduling policies

The scheduling policy decides which tasks that are to be executed and for how long by allocating the processors available in the system. The *runnable* tasks are placed in one or more *ready queues* waiting to be scheduled for execution. Tasks are removed from the ready queue when blocked by a system call or delayed until a specified relative or absolute time. Employing *preemptive* scheduling, a running task may be replaced by another runnable task and put back on the ready queue, while employing *non-preemptive* scheduling, tasks run until they are delayed, blocked by a system call or voluntarily *yield* the processor. This work only considers preemptive scheduling.

The most widely used policy is Fixed Priority Scheduling (FPS), where each task is given a fixed *priority* and resources are allocated to the task according to this priority. Usually tasks with the same priority are handled in first-in-first-out (FIFO) order, and the following rules apply:

- A blocked task that becomes ready is added at the tail of the ready queue for its active priority.
- When a task loses inherited priority, the task is added at the head of the ready queue for its new active priority.
- When a task executes a non-blocking delay statement or yield, it is added to the tail of the ready queue for its active priority.

- A running task is preempted whenever there is a nonempty ready queue with a higher priority and is then added at the head of the ready queue for its active priority.

Round-robin (RR) arbitration may be used instead of FIFO order to manage tasks with equal priorities. In this case the running task is moved to the end of the ready queue for its active priority after having been executed for a given time. Execution time clocks and timers may be used internally by the scheduler for an efficient and accurate implementation of this scheme [45]. While RR gives some degree of fairness between tasks of the same priority it does not improve schedulability, and also makes it harder to reason about the response time of tasks. Thus it is best suited for non-real-time tasks.

Another policy is Earliest Deadline First (EDF), where the processor is allocated to the task with least remaining time until its deadline. Thus EDF is a *dynamic* policy as priorities are decided and changed at run-time. EDF is optimal for uni-processor systems, allowing up to 100% utilization, but is more complex to implement and is also vulnerable to cascades of missed deadlines if tasks are allowed to continue executing after a deadline miss.

2.1.2 Scheduling analysis

Rate monotonic analysis (RMA) may be used to assign priorities to tasks and check the schedulability with FPS under the assumption that the processor is the only shared resource, that all tasks are periodic, that the deadline equals the next release, and that there is zero overhead of context switch between tasks [37].

With RMA each task i is assigned a priority according to their period T_i – the task with the shortest period is given the highest priority and so on. Note that when using RMA the priority of a task does not say anything about the *importance* of the task. Also, the required amount of execution time needed by task i is assumed to be a known constant C_i. It is assumed without further evidence here, that the task set is schedulable also if the tasks use less than C, thus the WCET or execution time budget may be used as C.

The utilization-based test is sufficient but not necessary as it gives the least upper bound (LUB) of the total processor utilization U for a task set with N tasks:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) = \text{LUB} \tag{2.1}$$

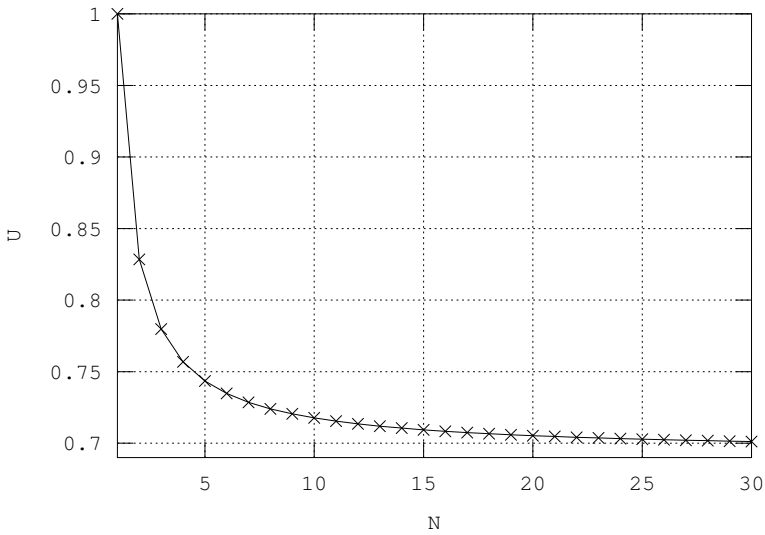

Figure 2.1: *The least upper bound (LUB) of processor utilization with rate monotonic analysis (RMA) for N tasks.*

As seen in Figure 2.1 the LUB asymptotically approaches 69.3% utilization when the number of tasks in the set increases towards infinity.

Response time analysis provides a sufficient *and necessary* test for schedulability. The test it will not be described formally here, and is instead performed by drawing Gantt charts. The test is done by releasing all tasks simultaneously at what is called the *critical instant*. If a task is schedulable when released at the critical instant it will also be schedulable at any other release configuration [37]. Only the time of the longest task period after the critical instance has to be considered. If all tasks reach their deadlines within this interval the system is schedulable.

Task	T	C
A	20	5
B	40	10
C	60	20

Table 2.1: *Example periodic task set. Priorities are* $p_A > p_B > p_C$.

An example task set is shown in Table 2.1. The total utilization for the task set is 83.3% and thus the utilization-based test fails as the LUB for $N = 3$ is 78%. However, this does not necessarily mean that the task set is not schedulable – as seen from the Gantt chart in Figure 2.2 all tasks reach their deadlines, and the task

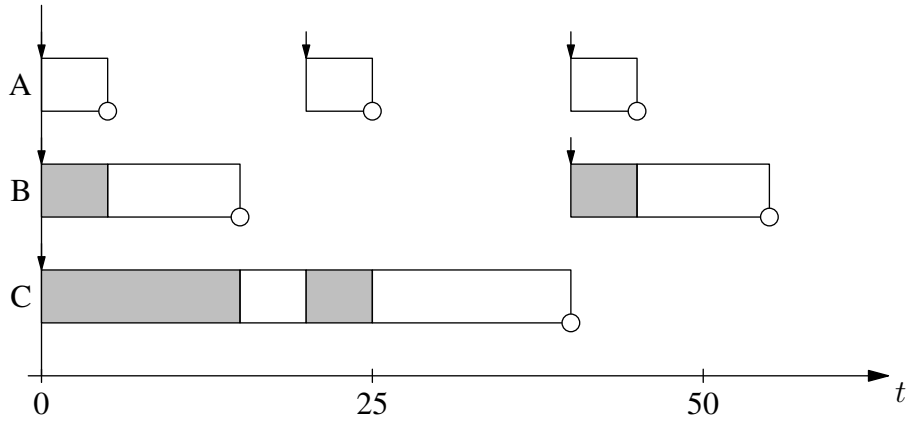

Figure 2.2: *Gantt chart for the example task set in Table 2.1. Task release is shown by a black arrow. Tasks are drawn as a white rectangle when running, and gray when another task preempts it. Completion before deadline is indicated by a white circle.*

set is indeed schedulable.

2.1.3 Shared resources

When using preemptive scheduling, shared resources must be protected with mutual exclusion. This can be done using explicit *semaphores* and protected regions of the code, but this primitive method is prone to programming faults that may cause a *deadlock* – a situation where several tasks all are blocked waiting for each other to release the shared resources. A better approach is to use a *monitor* that contains the shared resources and automatically takes care of the mutual exclusion.

It is also important to avoid *unbounded priority inversion* – a situation where a task is waiting for a lower priority task to release a shared resource, while this task again is preempted by a third task preventing it from finishing with the resource. This can be avoided by using a *priority inheritance* protocol with shared resources, the most practical being the *ceiling protocol* where a shared resource is given a fixed priority ceiling equal to or higher than the priority of all tasks accessing it. When a task acquires the resource it inherits the ceiling priority and thus no *unbounded* priority inversion can occur – the worst-case time waiting for a task with lower priority to release the shared resource is the highest WCET of all its operations.

Response time analysis is also possible for monitors using different priority inheritance protocols.

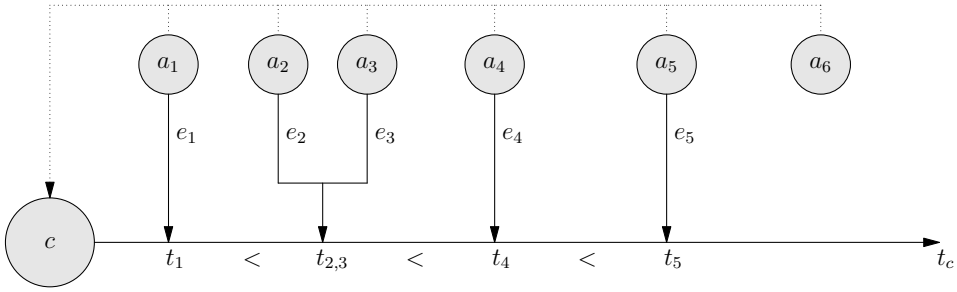

Figure 2.3: *Six alarms associated with one clock. All alarms but a_6 are set. Assuming that FIFO order is used and that a_2 was set before a_3 the events will be handled in the order e_1, e_2, e_3, e_4 and e_5.*

2.1.4 Timing services

A *clock* measures the passage of time using a physical process as reference, typically a crystal oscillator. Clocked time is discrete and represented by a count of ticks $c \in \mathbb{N}_0$. Each tick corresponds to a duration T, so the measured time is $t = T \cdot c$. Most clocks have inaccuracies caused by jitter and drift when compared to a reference clock. If the duration between the ticks is not a constant T but a stochastic function $\hat{T}$ it is called jitter. If the expected duration $E(\hat{T})$ between the ticks is not equal to T the clock will also drift compared to a reference clock. This drift will accumulate over time. Such inaccuracies are not considered in this work. Clocks are allowed to be stopped and resumed, but required to be monotonic. Therefore the following relation holds between samples where t_i denotes the i'th sample of a clock:

$$t_1 \leq t_2 \leq \ldots \leq t_{i-1} \leq t_i \quad \forall i \in \mathbb{N} \tag{2.2}$$

The real-time clock (RTC) is used for system operations such as task release and setting task deadlines. There is only one instance of this clock. The clock is activated at system start-up, also called its epoch, and is never stopped. Execution time clocks are used to measure the total time an executable entity has been running on the system. There is one clock for each entity. The clock is started when the entity is scheduled for execution and stopped when it is suspended, blocked, preempted by another entity, or terminated. In this work the executable entities considered are tasks and interrupt handlers.

An *alarm* is associated with a clock and is used to generate an *event* that occurs when the clock reaches a specified time t_e. An alarm is said to be *set* with a *handler* that is to be called when the event occurs, at a time $t \geq t_e$. Several alarms may

be associated with a single clock as seen in Figure 2.3. Event occurrences for one clock are required to be handled in order with earliest event first. The order for events occurring at the same time is not specified, but FIFO order will be used in this work unless stated otherwise.

2.1.5 Overrun handling

Different overrun handling policies exist to prevent deadline miss and system failure in the case of a task execution time overrun. Harbour et al. describes the following schemes [21]:

- *Handled*: The overrun is recorded and the task allowed to continue executing. This may be used for testing, for critical tasks that must be allowed to finish their work, or in cases where an occasional overrun is acceptable.
- *Stopped*: The task instance is stopped in the case of an overrun by the use of asynchronous control such as the abort statement in Ada, and is not repeated. The task starts executing normally the next time it is released.
- *Imprecise*: The task consists of a mandatory part that is usually short, and an optional part that refines the result. The optional part is aborted in case of an overrun. This scheme allows fixed priority for tasks executing algorithms where predicting the WCET is hard.
- *Lowered*: The task is lowered to background priority in the case of an overrun to avoid deadline miss for tasks with lower priority. The task may finish if there is sufficient CPU resources. The task priority is restored at the next release.

Another alternative is to reconfigure the system into a safe-state when an execution time overrun is detected. This is also possible when asynchronous task control is not available such as for the Ravenscar profile [17].

2.1.6 Execution time servers

Execution time servers allow a group of sporadic tasks with soft deadlines to be executed with higher priority than the periodic tasks with hard deadlines – thus giving low *average* response time for the sporadic tasks while guaranteeing a response time within the hard deadline for the periodic tasks. To set a budget for the sporadic tasks a *group budget* mechanism is needed.

The deferrable and sporadic server are the most used algorithms. Both allow the group of sporadic tasks to be modeled as a periodic task with period T when applying RMA. The deferrable server works by replenishing the group budget for the sporadic tasks registered with the server periodically with period T. If the budget is exhausted the tasks can either be halted by asynchronous task control, or given a priority below that of the periodic tasks to avoid further interference. When the budget is replenished the tasks are resumed or restored to the high priority. An appealing aspect of the deferrable server is that no knowledge of how the sporadic tasks execute is needed.

The sporadic server is quite similar to the deferrable server except that the consumed execution time for a sporadic task starting at time t is returned to the server at $t + T$. The sporadic server is much more complex than the deferrable, and the sporadic task must communicate with the server during execution. However, the benefit of the sporadic server is improved average response time.

2.2 The Ada programming language

In the 70s the U.S. Department of Defense ordered a programming language to replace the myriad of languages and dialects used for its different projects. A french team won the contract with the programming language Ada, named after Ada Lovelace – the daughter of the poet lord Byron and allegedly the world's first programmer, working on the mechanical Babbage machine.

The programming language was ISO-standardized in 1983 as Ada 83. There was a major revision of the language in 1995, known as Ada 95, bringing changes to tasking, added object-oriented programming (OOP) through *tagged types*, and more. The current revision is Ada 2005, which is an amendment to Ada 95, and brings Java-inspired improvements to the OOP-model such as supporting prefix method call notation and interfaces, more flexible access types, enhanced structure and visibility control for packages, extensions to the standard library, and new tasking and real-time features [8]. The coming Ada 2012 brings dynamic contracts, more flexible expressions, further extensions to the standard library, improved support for multi-processor system, and execution time measurement for interrupt handling [9].

Since Ada was designed for use in large high-integrity systems it has many safeguards against common programming faults. The language also has excellent support for development and maintenance of large applications by its notation of packages. Furthermore, Ada has language support for tasking and a rich set of

Listing 2.1: *Task type example.*

```
task type Worker
   (P : System.Priority;
    N : Character)
is
    pragma Priority (P);
end Worker;

task body Worker is
    Next : Time := Clock;
begin
    loop
        delay until Next;
        Put_Line ("Hello! My name is " & N & '.');
        Next := Next + Seconds (1);
    end loop;
end Worker;

A : Worker (Default_Priority + 2, 'A');
B : Worker (Default_Priority + 1, 'B');
```

synchronization primitives, making even multitasking applications portable.

2.2.1 Tasking features

Ada allows tasks to be specified either as a single instance, or as a task type with many possible instantiations that can be declared with *discriminants* as parameters. The body of the task defines the code that the task is to execute. An example defining a task type and two instances is shown in Listing 2.1. In this example the real-time package shown in Listing D.1 with its high-precision real-time clock is used for task delay.

Several real-time scheduling policies, or *dispatching policies*, are supported by Ada: FPS with FIFO within priorities or round-robin dispatching, EDF [13] and coherent mixes of these.

Protected objects, a type of monitors, provide mutual exclusive access to internal data through protected operations – procedures, functions and entries. Protected objects are also used for communication and synchronization between tasks, and

Listing 2.2: *Protected type implementation of mutex.*

```
protected type Mutex is
   entry Lock;
   procedure Unlock;
   function Is_Open return Boolean;
private
   Open : Boolean := True;
end Mutex;

protected body Mutex is

   entry Lock when Open is
   begin
      Open := False;
   end Lock;

   procedure Unlock is
   begin
      Open := True;
   end Unlock;

   function Is_Open return Boolean is
   begin
      return Open;
   end Is_Open;

end Mutex;
```

as interrupt routine handlers. As with tasks, a single instance may be defined or a protected type with many instances. Entries are associated with a *guard* condition which will block entry callers until the guard evaluates to true. This allows task synchronization through protected objects. How protected objects achieve mutual exclusion and queue tasks blocked on an entry is implementation dependent. Ada defines the priority ceiling protocol, and FIFO and priority queuing policies for entries.

An example of a protected object type used as a mutex is shown in Listing 2.2. The mutex is only used as example as its functionality is well known, otherwise it is meaningless to use a high-level synchronization primitive such as the protected object to implement a low-level primitive such as the mutex.

Full Ada tasking supports more complex features than shown in the example, such

as task rendezvous with entries and the select statement, allowing tasks to communicate directly, with the possibility of timeout for the task waiting for others; asynchronous abortion of code blocks; and the requeue statement to move a task to the queue of another entry. Since these features are not supported by the Ravenscar run-time environment they are not described further. The reader is referred to the excellent books by John Barnes [7] and Alan Burns and Andy Wellings [12, 15].

2.2.2 The Ravenscar profile

While Ada is much used within high-integrity systems, the concurrent constructs of the language have often been excluded as being non-deterministic and inefficient [11]. Instead methods such as the cyclic executive [5, 12] has been used. Advances in static analysis have made it possible to check hard deadlines when using preemptive fixed priority scheduling. This has led to development of the Ravenscar profile [10], a sub-set of the Ada tasking model designed to provide the static and deterministic environment needed to perform static analysis [11]. The simplicity of the tasking model also allows efficient run-time environments. The sequential parts of Ada are not affected by the profile [11].

The Ravenscar profile is specified as a set of configuration pragmas defining restrictions to the Ada tasking model and the required dynamic semantics [10, 32]. The following features are supported [11]:

- Tasks types and objects defined at library level.
- Protected types and objects, defined at library level, limited to one entry having a simple guard and a queue length of one.
- FIFO within priorities dispatching policy for tasks.
- Ceiling locking protocol for protected objects.
- The Ada.Real_Time package and the **delay until** statement.
- The Ada.Execution_Time and Ada.Execution_Time.Interrupts packages for execution time measurement of tasks and interrupt handling.
- Synchronous task control with suspension objects.
- Protected procedures as statically bounded interrupt handlers.

A static set of tasks and protected objects is achieved by only allowing such objects to be statically declared at library level and disallowing task termination. Dynamic attachment of interrupt handlers and dynamic change of task priorities

with the exception of changes caused by ceiling locking is also prohibited. Tasks may not have entries, thereby allowing task communication and synchronization only through protected objects or suspension objects. A protected object may have a single entry with a queue length of one using a simple barrier. The **requeue** statement and asynchronous control are disallowed. There can be no relative delay statements and the profile forces the use of the real-time package for timing purposes.

2.2.3 Timing events

Timing events allow protected procedures to be called at a specified time without the need for a task or delay statement to control their activation. The package used for timing events is defined as shown in Listing D.2.

A Timing_Event object is said to be *set* if associated with a non-null handler and *cleared* otherwise. The type Timing_Event_Handler identifies a protected procedure that will be executed when the timing event *occurs*. There are two procedures for setting a timing event with a handler, both named Set_Handler. One takes the absolute time of the event and the other uses relative time. If Set_Handler is called for an already set event, the handler is replaced. If called with a null handler the event is cleared. Handlers may be cancelled using Cancel_Handler which returns whether the handler was cancelled or not. The function Current_Handler returns the current handler of the event, while the function Time_Of_Event returns the time when the event will occur.

Implementations are required to perform operations on a timing event object atomically, and are also required to document the upper bound on the overhead of the handler being called.

2.2.4 Execution time control

The package Ada.Execution_Time shown in Listing D.3 defines the type CPU_Time which represents the elapsed execution time and the function Clock to get the execution time of a task [33]. The execution time of a task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf [33].

With Ada 2005 it was implementation defined which task, if any, that was charged the execution time used by interrupt handlers. Ada 2012 has the ability to account for either the total or separate execution time of interrupts handlers. The constant

Interrupt_Clocks_Supported indicates if the system supports measuring the total execution time of interrupt handlers by the use of the function Clock_For_Interrupts. The function will raise a Program_Error when called if this is not supported. The constant Separate_Interrupt_Clocks_Supported indicates whether or not the system supports measuring the execution time of interrupt handlers separately. This functionality is provided by the child package Interrupts shown in Listing D.4. In this child package the function Clock returns the execution time for the handler of the given interrupt or raises Program_Error if separate execution time for interrupts is not supported.

The child package Timers shown in Listing D.5 defines the tagged type Timer. An object of this type represents the source of an execution time event for a single task and is capable of detecting execution time overruns. As for timing events a timer is said to be set if associated with a non-null handler and cleared otherwise. All timers are initially cleared. The type Timer_Handler identifies a protected procedure to be executed when the timer *expires*. Timers are set and cancelled as with timing events with the exception of the absolute time for Set_Handler being given as CPU_Time. The function Time_Remaining replaces Time_Of_Event and returns the time remaining until the timer expires. Operations on a timer-object are required to be atomic. The number of timers possible for a single task is allowed to be limited by the implementation, and an exception Timer_Resource_Error is to be raised if this number of timers is exceeded.

The child package Group_Budgets shown in Listing D.6 allows execution time budgets to be set and replenished for a group of tasks. A user-provided handler is called when the budget has expired. Tasks may be added or removed from the group at any time, but a task can only belong one group at a time, and all tasks in a group has to be bound to the same CPU. Handlers are set and cancelled in the same way as timing events and timers.

2.2.5 The object-oriented real-time framework

The object-oriented real-time framework for Ada 2005 was initially designed and implemented by Andy Wellings and Alan Burns at the University of York [15, 60, 61], and has since become a de-facto standard within the real-time Ada research community. The framework has been extended with support for operating modes and mode changes [43], multi-processor systems using the coming Ada 2012 features [49], and execution time servers for interrupt handling using the non-standard interrupt execution time timers by the candidate [31].

The original framework consists of four major components [60]:

Listing 2.3: *The basic real-time task type of the framework.*

```
task body Simple_RT_Task is
begin
   S. Initialize ;
   loop
      R.Wait_For_Next_Release;
      S.Code;
   end loop;
end Simple_RT_Task;
```

1. The package Real_Time_Task_States defining the abstract tagged task state type that contains the task initialization code and the code to be executed at each release, the relative deadline, execution time budget, the priority of the task, and notification handlers for deadline miss and overruns. Child packages define periodic, sporadic and aperiodic abstract task states inheriting the abstract task state, each adding relevant parameters for the type of task.

2. The package Release_Mechanisms defining the synchronized interface for releasing tasks either as a result of the passage of timer or some event. Extended interfaces with overrun detection and deadline miss detection are also defined. Child packages implement periodic release mechanisms and one sporadic mechanism triggered by a procedure call with a minimal inter-release time (MIT), both types with and without overrun and deadline miss detection; and aperiodic release triggered with execution time servers. The release mechanisms use the attributes of their corresponding task states.

3. The package Real_Time_Tasks defining the task types that perform the work by dispatching calls to the provided release mechanism and task state. The simplest of the types shown in Listing 2.3 first initializes the object, and then waits for release and executes the code in an infinite loop. The two more advanced task types use asynchronous control to abort the task code in the case of deadline miss or execution time overrun, and then inform the task state by calling the corresponding handlers.

4. The package Execution_Servers defining the synchronized interface for execution time servers and generic parameters for this. Child packages implement the deferrable and sporadic server for use with aperiodic tasks.

Using this framework, the programmer needs only create a tagged task state type

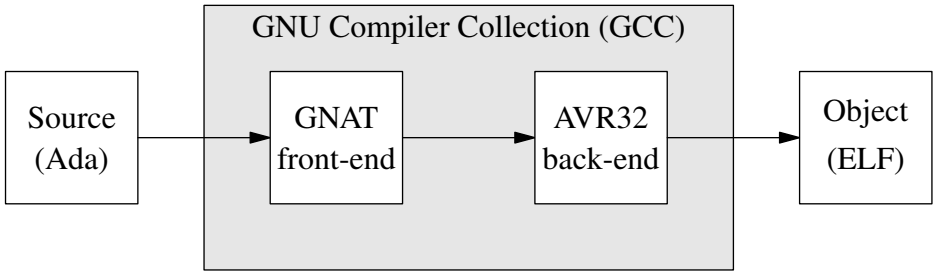

Figure 2.4: *The GNU Compiler Collection with front- and back-ends.*

inheriting the desired abstract type, and implement the abstract procedures for the tasks code and handlers. Objects of this task state can then be declared together with the appropriate release mechanism, real-time task type, and execution time server in the case of aperiodic tasks. Using the framework is easy as the boiler-plate parts are predefined, and also very flexible as the parts are interconnected through dispatching calls.

2.3 The GNU Ada Compiler (GNAT)

The GNU Compiler Collection (GCC) was released in 1987 by Richard M. Stallman, initially named the GNU C Compiler, and is the heart of the GNU Project maintained by the Free Software Foundation (FSF). Following the philosophy of FSF, GCC is open-source and "free as in liberty" – everyone can obtain the code, modify, compiler and redistribute it under the GNU Public License (GPL) as long as they do not deny this right to others [54].

GCC supports a great number of programming languages and computer architecture targets by a design as shown in Figure 2.4. It uses language *front-ends* that compile code into the internal tree representation, and target back-ends that create the assembler output from this and perform machine dependent optimization. The GNU Project also comes with assemblers, linkers and other tools in the GNU Tool-chain, the GNU Debugger (GDB) and more.

The GNU Ada Compiler (GNAT) is the GCC front-end for the Ada programming language. It was developed at the University of New York on contract with the U.S. Air Force and was originally called the GNU Ada Translator, hence the acronym GNAT. In a parallel project, the POSIX based real-time run-time library was developed at the University of Florida. After the completion of the project, the company AdaCore was established by project members to provide support for

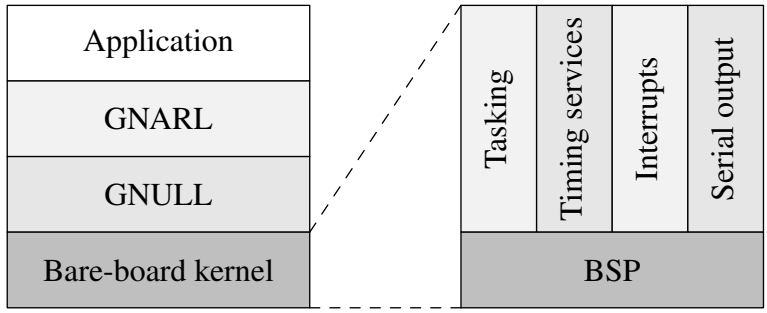

Figure 2.5: *The GNU Ada Library (GNARL). Here the GNU Low-level Library (GNULL) is implemented by the bare-board real-time kernel to the right.*

GNAT and further develop it into a full-featured industrial strength compiler [42]. AdaCore distributes GNAT both as the supported PRO version and the free/libre GPL version. The latest version is GNAT 2011 that brings supports for Ada 2012.

GNAT is more than just a language front-end; it also consists of the GNU Ada Run-time Library (GNARL), and Ada specific tools such as the binder, linker, builder with support for project files, and more, making it a complete development tool-chain for high-integrity systems.

2.3.1 Restricted GNARL

The GNU Ada Run-time Library (GNARL) seen in Figure 2.5 implements the features of Ada not directly supported by the compiler such as tasking, interrupt handling, standard libraries, distributed programming, system interfaces, and more. The routines of GNARL are called by the application either directly or indirectly by compiler generated code.

The restricted Ravenscar version of GNARL is carefully designed to take advantage of the simplifications allowed by the profile [47]. Task management is simplified since all tasks are defined at library level, cannot terminate, and have fixed base priority. All task data structures are statically allocated, thus memory requirements are determined at link time. Protected objects are simplified since there are no asynchronous operations, no time-out on entry calls and no varying queue length on entries. On single processor systems mutual exclusion is ensured by the ceiling priority protocol and scheduling policy. Evaluation of protected entries may be done by proxy, thereby improving performance by reducing the number of context switches [47].

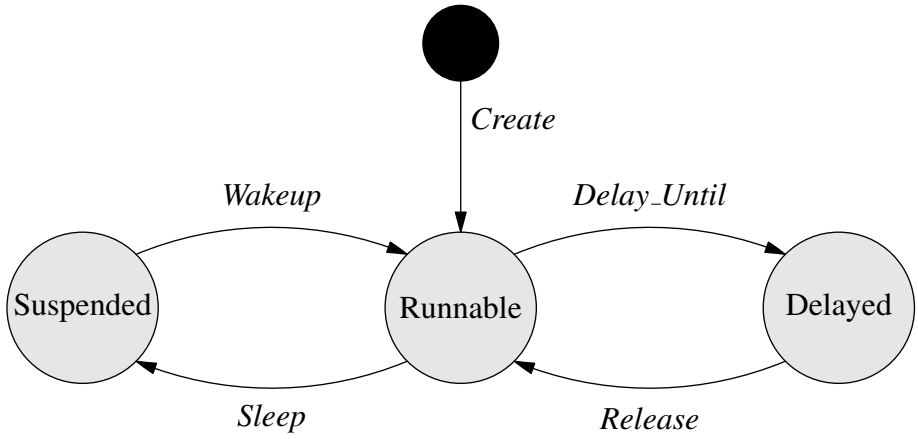

Figure 2.6: *Thread states of the bare-board real-time core: Runnable when in the ready queue; Suspended when unconditionally blocked (on an entry); and Delayed when blocked until a specified absolute time.*

2.3.2 The bare-board Ravenscar run-time environment

For most systems the GNU Low-Level Library (GNULL) is a translation layer between generic and actual operating system calls, but for the bare-board run-time environment it fully implements the needed dynamic semantics of the Ravenscar profile as to the right of Figure 2.5. This is done by including a multi-tasking core based on the Open Ravenscar Kernel (ORK) [18, 48]. This kernel was developed at the Polytechnical University of Madrid (UPM) on contract with the European Space Agency (ESA) for use on the ERC32 and LEON space application processors, and was later integrated into GNARL by José F. Ruiz at AdaCore [47].

The core implements preemptive fixed priority scheduling with ceiling locking, having 256 priorities including the interrupt priorities. However, the total number of priorities can easily be changed if needed. To allow interrupt nesting whilst avoiding priority inversion, each interrupt priority has its own interrupt stack [47]. Interrupts are masked as long as the running task has higher or equal priority to the given interrupt. All interrupts are masked while modifying core data in order to ensure mutual exclusion.

The *thread* type used for tasking is very simple due to the limitations of the Ravenscar profile and only has three states as seen in Figure 2.6. All thread operations on runnable threads are done on the *running* thread first in the ready queue. The queue is only modified as a result of threads being inserted, removed, and when the priority of the running thread is changed according to the FIFO within priori-

ties dispatching policy. Whenever the first thread in the queue is not equal to the running thread a *context switch* will take place before leaving the core. There is no idle thread in the run-time environment. Instead the thread that finds the ready queue empty when leaving the kernel is reinserted into the queue and enters an idle-loop waiting for any thread to be made runnable by an interrupt. The thread only leaves the idle-loop when itself or another thread is runnable again. Threads enter the idle-loop one at the time.

The timing services of the core provides as high precision as possible while supporting the 50 year time span required by the Ada standard [33]. This is done by using a 64-bit value for time divided into two parts. The least significant part is present in the system timer called the *clock*. The most significant part is stored in memory and is incremented every time the clock timer overflows. A second timer, the *alarm*, is used in one-shot mode to release delayed tasks with high-precision between the clock overflows.

The multitasking core builds on a device specific Board Support Package (BSP) consisting of peripheral drivers, CPU primitives and assembler files facilitating context switch, low-level interrupt handling and interrupt masking.

2.4 The Atmel AVR32 architecture

The Atmel AVR32 [3] was designed at Atmel Norway in close cooperation with NTNU, and is a 32-bit RISC architecture optimized for high code density and high computational throughput with low power consumption [57].

The architecture has a fairly small register file consisting of 13 general purpose registers (named R0 to R12), the link register (LR) used for storing the routine return address, the program counter (PC) and the system register (SR). Instructions are both 16- and 32-bit wide for higher code density. Being a RISC architecture, the AVR32 has a number of load / store operations, many with the possibility to increment or decrement the pointer register before or after memory access. There are also instructions to load or store multiple registers.

The AVR32 has four interrupt levels, and exceptions such as the Non-Maskable Interrupt (NMI) and illegal address exception. The entry point for each of the four levels is configurable, and registers R8 to R12, LR, SR and PC are automatically stored before entering the handler. For the AVR32A microarchitecture the registers are stored on the system stack, while for the high-performance AVR32B microarchitecture shadow register files are used.

Interrupts are managed by the *interrupt controller* that groups the interrupts of the specific part, and allows each such interrupt group to be assigned to any interrupt level.

2.4.1 The UC3 microcontroller series

The UC3 microcontroller series [4] is the second implementation of the AVR32 architecture, and the first of the AVR32A microarchitecture. It is primarily intended for embedded control applications where deterministic execution time is paramount. The UC3 implements the DSP instructions of the AVR32 architecture set such as several single-cycle multiply and accumulate instructions for both modular and saturated arithmetic.

The UC3 has a three-stage hazard-free pipeline consisting of:

1. The instruction fetch stage optimized for on-chip Flash memory.
2. The decode stage that decodes the instructions and sets up control signals.
3. The execution stage consisting of the ALU preforming arithmetic and logical operations, the multiplication unit preforming multiply and multiply-accumulate operations, and the load / store unit accessing the SRAM and high-speed bus.

An important feature of the UC3 is the internal SRAM integrated with the CPU pipeline. The system bus is bypassed, allowing deterministic, single-cycle read-/write memory access. A high-speed bus (HSB) slave interface to the SRAM allows DMA controllers or other HSB masters to directly write or read data from memory. Arbitration with a programmable priority scheme is performed if the CPU and a high speed slave request access simultaneously.

2.4.2 Hardware timers

The 32-bit COUNT / COMPARE system registers of the Atmel AVR32 architecture are used to count the number of elapsed CPU cycles and allow an interrupt to be triggered. The COUNT register is reset to zero at system start-up and is incremented by one every CPU clock cycle. The COMPARE interrupt is triggered when COUNT equals COMPARE, cleared when COMPARE is written, and disabled when COMPARE is zero, which is also the reset value of the register. For newer UC3 revisions the COUNT register is reset on COMPARE match. It is however possible to disable this behavior in the CPU configuration register.

The Timer / Counter peripheral of the UC3 may be used for signal waveform generation and measurement, and as an one-shot or periodical timer generating interrupts. The peripheral has three 16-bit channels with selectable clock source and prescaling.

Chapter 3

Main contributions

3.1 GNAT for AVR32

Article A.1 is titled *"An efficient and deterministic multi-tasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller"* [26] and describes how the GNU Ada Compiler (GNAT) was ported to the Atmel AVR32 architecture and how the GNAT bare-board run-time environment was ported from the LEON architecture to the UC3 series of microcontrollers. This work made Ada available for the AVR32 for the first time, and provided a solid research platform for Ada real-time systems.

Refer to Appendix C for details about how the source code of the compiler and run-time environment is obtained, configured and build.

3.1.1 Porting the GNAT front-end to the AVR32 GCC back-end

The GCC back-end for the AVR32 architecture was initially developed at NTNU and is now maintained by Atmel Norway. It is not yet in the official code distribution of GCC, but patches for GCC 4.3 are available at Atmels web-pages.

Since both the front-end and back-end are open source components of GCC, porting GNAT to the AVR32 was mainly a matter of applying the GNAT patches from AdaCore to the already patched GCC source code provided by Atmel. Due to changes to the source code by the AVR32 patches, some of the GNAT patches failed to apply and needed to be fixed manually.

The ported compiler may also be used together with AVR32 Linux on the AVR32 AP7 series of application processors [2]. In this case the standard POSIX based run-time environment should be used. This work was not prioritized as Linux was not found suitable for the research and the AP7 has now become deprecated.

3.1.2 Porting the run-time environment

When porting the run-time environment to the UC3, the board support package (BSP) including the context switch routine, low-level interrupt handler and peripheral drivers needed to be rewritten due to differences in hardware.

The context switch routine for the UC3 could be written in as few as 15 instructions with no branches due to the small register file of the AVR32. This was a significant reduction compared to the LEON architecture that has a large register file with register windows which means that the execution time of the routine will vary depending on the current window size. The UC3 context switch has a constant execution time which eases execution time analysis of applications.

The AVR32 has a peripheral interrupt controller that groups interrupts. Each group is configured to one of the architectures four interrupt levels by software, and there is a low-level handler for each level. The interrupt ID is found by first reading the interrupt cause register of the level to find the interrupt group, and then the interrupt request register of that group to find the interrupt line.

The timing services uses two hardware timers named *clock* and *alarm*. On the AVR32 two channels of the 16-bit Timer / Counter peripheral are used for these timers. The clock timer keeps the least significant part (LSP) of the system clock. On overflow, the most significant part (MSP) of the clock that resides in system memory, is incremented by the clock interrupt handler. The alarm timer is used in one-shot allowing fine grained task release within the clock periods.

3.2 Implementation of Ada 2005 execution time control

Article A.4 is titled *"Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture"* [30] and describes how the new Ada 2005 timing event and execution time control features were implemented for the GNAT bare-board run-time environment on the Atmel AVR32 UC3. A novel feature of this implementation is that the execution time for interrupt levels are measured separately, and there is an interrupt timer for all levels except the highest

one. This increases the accuracy of execution time measurement for tasks and allows execution time control for interrupts for the first time.

3.2.1 Timing events

Prior to implementing Ada 2005 timing events the alarm timer was used exclusively for waking up delayed tasks. The functionality needed for this was spread over three kernel packages. When implementing timing events it was decided to use the same alarm mechanism both for waking up tasks and handling timing events and to gather all the functionality needed in System.BB.Time. By using the same mechanism both for timing events and waking up delayed tasks, a clean implementation was achieved. Furthermore, lower overhead may be achieved as a timing event and a task release occurring at the same time are handled by the same alarm interrupt.

Internally, alarms are organized as a queue in ascending order by the timeout value of the events with a sentinel at the end. The queue is implemented as a doubly linked list in order for alarms to be quickly extracted if cancelled. The sentinel was added to simplify the code by removing the special case of having an empty queue. When inserting an alarm the queue is searched from the front to the end. The one-shot alarm timer is reprogrammed whenever the first alarm is changed. When the alarm timer interrupt is triggered, all alarms with timeout less than or equal to the current time are called and removed from the queue by the interrupt handler.

3.2.2 Execution time control

The execution time control features were implemented in a new package named System.BB.TMU where TMU stands for *Time Management Unit*. The package defines the type CPU_Time representing elapsed execution time as a 64-bit modular integer, and the timer mechanism that is used for execution time control combining a clock and a single alarm in one type. This is done as implementations are allowed to limit the number of timers for tasks. As recommended for the Ravenscar profile [17], the implementation allows for at most one timer for each execution time clock.

The system measures execution time as the number of CPU clock cycles used by a task since its activation by using the COUNT / COMPARE registers of the AVR32 architecture. The Base_Time of a timer is initially zero and is used for keeping track of absolute execution time. The execution time for an active timer is the sum

of the base time and the value of the COUNT system register. When a timer is deactivated its base time is incremented with the value of the COUNT register, thus the execution time of inactive tasks equals the base time. The correctness of the execution time measurement depends on the CPU counter never overflowing. To prevent this the value written to COMPARE is never greater than a constant C_{max} set to $2^{31}-1$. When COUNT equals this value a COMPARE interrupt will be pending causing the timer to be inactivated and its base time updated when handled.

Accuracy of task execution time measurement is improved by charging the execution time of interrupt handlers to the clock of a pseudo interrupt task for the corresponding interrupt level. This allows task budgets to be tighter and therefore allow for a higher utilization of the processor. Even more important, it allows the use of execution time timers for interrupt handling in order to set budgets and handle overruns. This allows protection against faults in interrupt generation or handling, for instance by blocking the handling of an interrupt when its budget is exceeded. Interrupt timers are not allowed for the highest interrupt priority since the kernel interrupts are of this level, and blocking these will result in system malfunction.

Execution by proxy improves system performance by reducing the number of context switches needed. When task τ_a executes a protected operation releasing task τ_b that is blocked on an entry, τ_a will also execute the entry on behalf of τ_b. By charging τ_b the execution time spent on the entry this implementation improves the accuracy of the execution time measurement and the execution time of the entry does not have to be added to the budget of τ_a. The usefulness of this feature is highly dependent on the implementation overhead for changing timers compared to the execution time of the entry. Testing of the implementation indicates that the overhead of changing clocks may be larger than the execution time of the entry. Still there is some usefulness in exchanging a variable overhead of entry-by-proxy execution with a constant.

3.2.3 Modifications to Ada 2005 standard library

In order to support execution time control for interrupts, additions had to be made to Annex D of the Ada 2005 standard [32] as shown in Listing 3.1. These changes were made in the existing execution time control packages specified instead of adding new packages to the standard library. In Ada.Execution_Time the function Interrupt_Clock was added to support execution time measurement for interrupt priorities. It returns the total time spent by all interrupt handlers of the given interrupt priority since start-up.

Also, the tagged type Interrupt_Timer was added to Ada.Execution_Time.Timers to

Listing 3.1: *First API proposal for interrupt execution time control.*

```
package Ada.Execution_Time is
   ...
   function Interrupt_Clock ( Priority   : System.Interrupt_Priority )
      return CPU_Time;
   ...
end Ada.Execution_Time;

package Ada.Execution_Time.Timers is
   ...
   Pseudo_Task_Id : aliased constant Ada.Task_Identification.Task_Id
      := Ada.Task_Identification.Null_Task_Id;

   type Interrupt_Timer ( I  : System.Interrupt_Priority )
      is new Timer (Pseudo_Task_Id'Access) with private;

end Ada.Execution_Time.Timers;
```

support execution time timers for interrupt priorities as shown in Listing 3.1. The type inherits Timer and takes the interrupt priority as discriminant. None of the operations of Timer are overridden as it is assumed that the same underlying mechanism will be used both for task and interrupt timers and that the only reason for having a separate type for interrupt timers is the difference in the discriminant.

3.3 Usage of execution time control in Ada

Article A.2 is titled *"A real-time framework for Ada 2005 and the Ravenscar profile"* [28] and describes how an object-oriented real-time framework for Ada 2005 [60] was adapted to the limitations of the Ravenscar profile. The framework is extended using the novel features described in Article A.4 to support execution time control for interrupt handling. The article also describes an example application demonstrating the use the framework.

3.3.1 Task states and release mechanisms

While the original framework takes advantage of the full Ada 2005 tasking model, the described framework cannot use mechanisms such as asynchronous control,

requeue and select statements, and dynamic priorities. Instead it has to rely on tasks voluntarily giving up the processor within the specified recovery time after an overrun is signaled by calling the procedure Overrun of the task state. If the task has not stopped within the recovery time the overrun handler is called again and some emergency action should be taken.

It is not specified in which way overruns are to be handled by the task state, but flag polling seems to be the most viable approach. Arguably this is not an elegant solution, but the options are limited without asynchronous control. The example applications have a periodic task state with an overrun flag that is polled in the loop performing the work of the task. The task aborts its work when the flag is set by the handler.

3.3.2 Extensions for interrupt handling

Flexible interrupt handling is supported by the framework by specifying an interface for interrupt states and an underlying interrupt handling mechanism, similar to the task state and underlying real-time tasks. However, the sporadic interrupt release mechanism is also implemented as in the original framework since this is more efficient for interrupts that simply release tasks.

An interface Interrupt_Server is defined for controlling the execution time of interrupt levels. This interface is implemented by the deferrable interrupt server that replenishes the execution time budget periodically and blocks all registered interrupts of its interrupt level if the budget is overrun. This allows applications to limit the execution time spent on interrupts and protect against burst of interrupts that would otherwise cause tasks to miss their deadlines. This functionality depends on the special interrupt level execution time timers implemented on the GNATforAVR32 run-time environment as described in article A.4 and is therefore architecture specific.

3.4 IRTAW-14 and ISO standardization

Article A.3 is titled *"Execution time control for interrupt handling"* [29] and was presented at the 14th International Real-Time Ada Workshop (IRTAW-14) held in Portovenere, Italy, 7 to 9 October 2009. The article is a sub-set of article A.4, presenting execution time control for interrupts by adding execution time clocks and timers for interrupt levels as shown in Listing 3.1. It proposes that these features should be added to the next revision of the ISO standard for the Ada programming

Listing 3.2: *Updated API proposal for interrupt execution time control.*

```
package Ada.Execution_Time.Interrupts is
   function Clock (I : Ada.Interrupts.Interrupt_ID) return CPU_Time;
private
   ...
end Ada.Execution_Time.Interrupts;

package Ada.Execution_Time.Timers.Interrupts is

   type Timer (I : Ada.Interrupts.Interrupt_ID)
      is new Ada.Execution_Time.Timers.Timer
     (Ada.Task_Identification.Null_Task_Id'Access) with private;

private
   ...
end Ada.Execution_Time.Timers.Interrupts;
```

language. The presentation of execution time control for interrupts given by the candidate is included in Appendix B.

3.4.1 Updated API proposal using interrupt ID

The reason for using execution time clocks for each interrupt priority instead of using interrupt IDs was primarily ease of implementation, efficiency and to reduce system requirements. Typically there will be many more interrupts than interrupt levels, which means that using IDs will require more memory. Also, the interrupt level is known when the system initiates the interrupt handler, while getting the interrupt ID often requires queries to a peripheral interrupt controller. This means that one can switch clocks earlier using interrupt priorities, which reduces the inaccuracy to the tasks execution time clock. Moreover, using interrupt priorities allows for automatic switching to interrupt clocks in hardware such as done by the earlier TMU design at NTNU [19, 52].

However, the use of interrupt levels instead of interrupt IDs raises some issues. On multi-processor systems several interrupts of the same level may be handled at the same time. This could be solved by having a separate set of interrupt level clocks for each CPU, but this is not an elegant solution. Also, knowing the execution time spent handling each interrupt gives better control than just knowing the

execution time spent handling interrupts of a given priority. After some discussion the workshop agreed that the candidates alternative proposal of interrupt clocks for each interrupt ID was preferable, and that the new functionality should be separated in new packages. The candidate presented an updated API proposal shown in Listing 3.2. The slides from this presentation are also found in Appendix B.

3.4.2 Other API proposals and workshop decision

At the workshop, the developers of the MaRTE run-time environment presented a solution of measuring the combined execution time of all interrupt handling using a single clock [46]. This gives the same benefits in increased accuracy of execution time measurement for tasks, but does not allow for interrupt execution time control. Using only one clock for interrupt handling with no timer also allows for more efficient implementation with low overhead.

The workshop decided to suggest execution time measurement both for separate interrupt IDs as presented by the candidate and all interrupts combined to be added to Ada 2012 [40, 58]. These features are now included in the working draft for the Ada 2012 standard [33] with some minor additions to check for run-time environment support as seen in Listing D.3 and D.4. However, interrupt timers allowing full execution time control for interrupt handling were deferred being viewed as too experimental, and are not available for Ada 2012.

3.5 Implementation of Ada 2012 execution time control

Article A.6 is titled *"Implementation and usage of the new Ada 2012 execution time control features"* [31] and describes how the run-time environment was updated to support the new execution time control features in Ada 2012. Execution time control for interrupts is now on Interrupt_ID basis instead of interrupt priorities as in article A.4. Furthermore, the real-time and execution time features use the same clock and alarm abstraction, reducing the amount of code needed for the implementation. This design also allows a single hardware timer to support these features, freeing other timer hardware for application use. Clock measurement is tick-less, removing the periodic clock overflow interrupts. Performance tests are done to find the additional overhead to context switches and interrupt handling caused by execution time control.

The interrupt timer is not included in Ada 2012, only interrupt clocks. The article describes the implementation of the interrupt timer updated to use Interrupt_ID and

the updated real-time framework extensions facilitating full execution time control for interrupt handling. An example application using this feature is given to demonstrate why interrupt timers should be included in the next revision of Ada.

3.5.1 Design and implementation

The functionality of the real-time clock (RTC) and execution time clocks are quite similar: both clocks support high accuracy measurement of the monotonic passing of time since an epoch, and both support calling a protected handler when a given timeout time is reached. The main difference is that the RTC is always active, while an execution time clock is only active when its corresponding task or interrupt is executed. The similarities allow a design where one implementation of clocks and alarms provides support for both execution time control and the real-time features. These types are defined in the internal bare-board run-time environment package System.BB.Time as shown in Listing D.7.

The package body declares the RTC, interrupt clocks, and the internal idle clock used when the system is executing the idle-loop. In order to save memory there is a pool of interrupt clocks and a look-up table with Interrupt_ID as index, instead of having a Clock_Descriptor for every interrupt. This is done since the bare-board Ravenscar run-time environment does not use dynamic memory in the kernel. Alternatively, memory for the interrupt clocks can be allocated on the heap when interrupts are registered. As before, the thread's execution time clock is stored in the thread descriptor type defined in the package System.BB.Threads. This type also has an alarm used for real-time delay.

After initialization of the package there are precisely two active clocks: the RTC that is always active and the ETC that points either to the execution time clock of the running thread, to the clock for the interrupt being handled, or to the non-visible idle clock. The ETC is changed as a result of a *context switch* between tasks; through *interrupt handling* where the interrupted execution time clock is pushed onto the stack and is later reactivated when the handler has been called; or by *system idling* where the idle clock is activated while the task executes the idle loop. To support the latter the task descriptor has an access named Active_Clock that points to the idle clock when executing the idle loop and the task's execution time clock otherwise. Execution time measurement does not take the effects of entry by proxy execution as in Article A.4 in consideration anymore since the overhead of doing this was deemed higher than the benefits.

The 32-bit COUNT / COMPARE registers of the AVR32 architecture are still used, but now both for execution time control and the real-time clock. In order to do

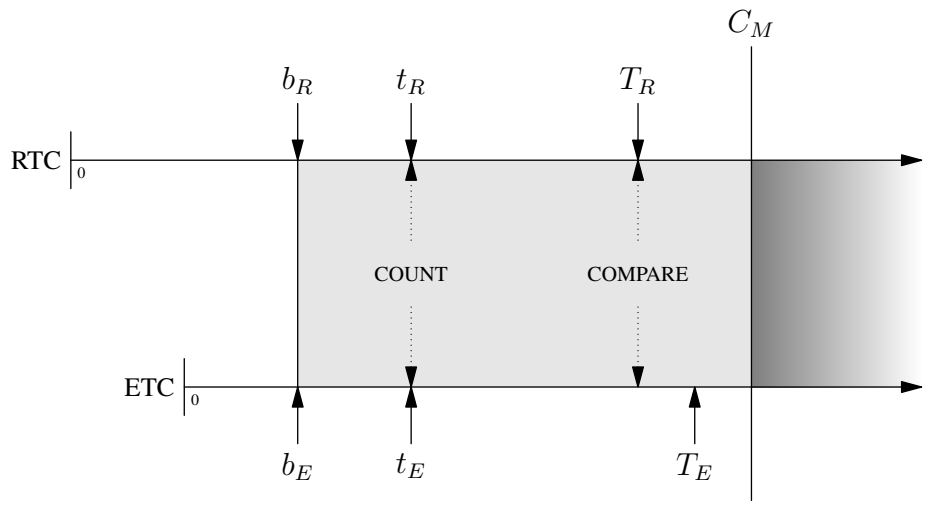

Figure 3.1: *Relation between the RTC and ETC, and the hardware timer registers. The base time of the clocks are here aligned, and the safety region is shown in dark gray.*

this the COUNT value is reset every time the ETC is changed. The base times of both clock are updated, and COMPARE is set according to the closest alarm of the RTC and the new ETC with the same safety region to prevent overflows. The relation between these variables and registers is shown in Figure 3.1. The COMPARE interrupt handler is called on COMPARE match and calls the handler of all expired alarms for the RTC and the interrupted execution time clock found on top of the stack.

3.5.2 Interrupt timer

To allow execution time control for interrupt handling, the proposed child package shown in Listing 3.3 defines the tagged type Interrupt_Timer that inherits from Timer. No body is needed for this package as all operations are inherited. The initialization procedure for timers checks if the object is of type Interrupt_Timer in which case it uses the interrupt clock instead of the task clock. Interrupt timers are used in the exact same way as task timers.

The interrupt timer is not a part of the Ada 2012 standard. It should however be considered added to the next revision as it provides execution time control for interrupts similar to that for tasks. If we measure the execution time for interrupts it should also be controllable by means such as the framework extensions described in Article A.6. This is important as the execution time spent handling interrupts

Listing 3.3: *Interrupt timer specification*

```
package Ada.Execution_Time.Interrupts.Timers is

   type Interrupt_Timer (I : Ada.Interrupts.Interrupt_ID)
      is new Ada.Execution_Time.Timers.Timer
     (Ada.Task_Identification.Null_Task_Id'Access)
     with private;

private

   type Interrupt_Timer (I : Ada.Interrupts.Interrupt_ID)
      is new Ada.Execution_Time.Timers.Timer
     (Ada.Task_Identification.Null_Task_Id'Access)
     with null record;

end Ada.Execution_Time.Interrupts.Timers;
```

may be very hard to predict, since the interrupts may be generated by external hardware that are not directly controlled by the application. Alternatives to interrupt timers are to count the number of interrupts and disable the interrupt if the count gets too high, or to poll the execution time of the interrupt after the handler is called and disable the interrupt if the budget is exceeded. These solutions are less precise and also less efficient than using interrupt timers.

3.5.3 Framework extensions

The framework components related to interrupt handling can be separated into three parts: (1) the interface Interrupt_Controller used to control hardware interrupt generation; (2) the protected interface Interrupt_Server used to control the execution time spent handling a given Interrupt_ID in accordance with some policy; and (3) the protected interrupt handlers of the application, the framework already provides the release mechanism Sporadic_Interrupt to release tasks as a result of an interrupt.

The interrupt timer was used to extend the object-oriented real-time framework to also provide execution time servers for interrupts following the same pattern as used for task execution time servers. While the task server controls the execution time for a group of tasks released sporadically, the interrupt server controls the execution time spent invoking one interrupt handler many times. The object-oriented

nature of the framework allows the creation of servers suitable for different needs. The deferrable server was implemented under the assumption that it is acceptable to ignore interrupts for a while. Another scheme may be to reconfigure the system into fail-safe mode in the case of interrupt overruns.

The deferrable interrupt server has a budget that is replenished periodically, and disables generation of the interrupt it controls if this budget is exceeded. Since there is no way to cancel the interrupt being handled in Ada, the budget has to allow for an overrun of one additional handler invocation for the cases where the budget is exceeded right after entering the low-level handler. A possible future enhancement could be to add a user handler that is called to notify the application when an interrupt is disabled, to allow for hardware diagnostics or other application dependent handling.

3.5.4 Example application

To demonstrate the usage of interrupt timers and the extensions to the real-time framework, a simple example application was developed. In this application the USART RX interrupt is used to read data received on the serial line. For this example, this is reasonable and efficient since *intended* usage of the system is that the characters are sent by the user typing in a serial communication program, which will limit the rate of interrupts. However, the high baud rate of the USART line means that the system could be overloaded with interrupts if this limitation is not respected. In turn this could cause the periodic real-time task in the example application to miss its deadline.

By using the deferrable interrupt server of the real-time framework, one can easily set a budget for the interrupt so that the real-time task is guaranteed sufficient execution time to meet its deadline. The server has the same replenishing period as the release period of the real-time task, allowing both events to be handled by the same RTC alarm handler. The severs budget is set so that the system is known to be schedulable using RMA. To test the interrupt server, data was also sent on the USART at full baud rate. It was observed that the USART interrupt was disabled when its budget was overrun and re-enabled when it was replenished. No deadlines were lost due to bursts of interrupts when the application was tested with the deferrable server, while several deadlines were lost during the bursts when the server was not used. This gives a good indication that the deferrable interrupt server works as intended.

The example application is typical in that we must assume a particular rate of interrupts, but cannot guarantee it as the generation of the interrupts is not controlled

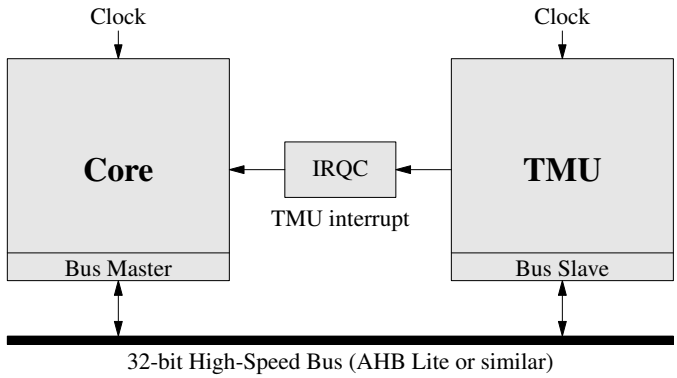

Figure 3.2: *Core and TMU connected to the HSB.*

by the application. Bursts of interrupts may also be caused by permanent or transient hardware faults. The result is that the system has to handle more interrupts than the budget allows for in the real-time analysis, if the effects of interrupt handling were analyzed at all, causing deadlines to be missed. The interrupt timers and extensions to the real-time framework provide an easy and robust way to protect real-time applications against these situations.

3.6 The hardware Time Management Unit (TMU)

Article A.5 is titled *"Functional specification of a Time Management Unit"* [24] and describes a dedicated hardware Time Management Unit (TMU) designed to improve the performance of execution time control. The TMU is designed for efficient memory-mapped access through the high-speed bus (HSB) of the microcontroller. The simplicity of the TMU design allows it to be added to existing System-on-Chip (SoC) designs with minimal effort.

3.6.1 Functional specification

The TMU was designed as a memory-mapped device accessible through the HSB as shown in Figure 3.2. The bus address and data are both assumed to be 32-bit wide. In addition to bus interface the TMU has a clock signal as input. The clock source of the TMU need not be the same as that used by the core. The TMU generates an interrupt signal that will usually be routed to the core through an interrupt controller.

Internally the TMU has a 64-bit COUNT register that is incremented on every positive edge of the clock signal. After COUNT is incremented it is compared with the 64-bit COMPARE register. If COUNT $\geq$ COMPARE then the interrupt signal is asserted. In order to atomically swap a new set of COUNT / COMPARE values with the current, two swap registers are provided. The registers are swapped when the final word of the swap registers is written, and the previous values of COUNT and COMPARE may then be read back. The swap registers allow for simple and efficient change of execution time clocks.

The COUNT register may also be accessed directly. When reading the high-word of COUNT the low-word is stored in an internal 32-bit buffer, and this buffered value is returned when the low-word is read. Similarly, the high-word value is buffered when writing the high-word of COUNT and the whole register is updated when the low-word subsequently is written using the buffered high-word and the provided low-word. Due to the buffering care must be taken not to interleave writing and reading of COUNT. If available it is recommended to use double-word load / store instructions so that the registers are read and written atomically.

The functional specification of the TMU is given in SystemC, a C++ library for high-level modeling and simulation of SoC designs. The specification is therefore executable, and can be integrated with other SystemC components for simulation in a larger system. The TMU was simulated with a minimal core running a set of tasks with FPS scheduling to verify the design.

3.6.2 Design rationale

By using 64-bit COUNT and COMPARE registers the absolute execution time values can be stored in registers instead of storing relative values as done when using 32-bit registers. Therefore, there is no need for translating the absolute execution time values used by applications into relative values loaded into the registers. Also by having a less-or-equal comparison instead of only equality it is no longer necessary to check if COMPARE is less than COUNT before setting the registers. Even more important is the ability of the TMU to swap COUNT / COMPARE registers atomically. This simplifies the change of execution time clocks in the context switch routine and low-level interrupt handlers. The design provides a simple, yet highly efficient hardware mechanism for implementing execution time control that leaves the policy entirely for the software. This simplifies the hardware implementation, and is also more flexible as the usage of the TMU is decided by software.

In contrast, in an earlier design at NTNU [52] that was implemented for the LEON architecture, the execution time of interrupts was measured automatically in hard-

ware by changing to the correct interrupt clock before the processor started handling an interrupt [19]. This design also supported blocking the interrupt in hardware after the deferrable server pattern. While the benefit of this design is zero overhead to interrupt handling, it is costly to implement and also limits the choice of execution time control policy to one predefined in hardware.

Also, the earlier TMU design had one interrupt clock for each interrupt level. This worked well for the LEON architecture that has one level for each interrupt, but would not work on the AVR32 architecture where the level of the interrupt groups are configurable, and several interrupts may be of the same level. The pending interrupt is found in the low-level interrupt handler, and the interrupt clock cannot be activated before this. However, if relative register values were used it would be possible to reset it before it is known which clock to activate. The early TMU design would also work well with the originally proposal for execution time control scheme where interrupt levels and not interrupt IDs were used.

3.6.3 Implementation for the UC3 microcontroller series

The above given TMU design was implemented and tested for the Atmel AVR32 UC3 microcontroller series by master student Stian Søvik at NTNU in cooperation with Atmel Norway [53]. This work was supervised by Amund Skavhaug and guided by the candidate.

When the TMU was implemented for the UC3, some technical changes were needed [53]. The unit was moved away from the high-speed bus to the peripheral bus to ease the implementation, and the clock signal driving the TMU was bound to the clock of the peripheral bus to allow for a synchronous design. This resulted in slightly longer access time and limited the frequency of the TMU clock to that of the peripheral bus, which may be lower than the CPU frequency.

Several registers were added to the interface of the TMU as shown in Table 3.1 to make it more like other UC3 peripherals and usable for a wider range of purposes [53]. A control register was added for enabling and disabling the TMU. The peripheral unit is now disabled by default in order to save power. Even though the COUNT register is 64-bit and not expected to overflow with the usage intended when the TMU was designed, an overflow interrupt was added to allow for other usages. Also, registers for getting the interrupt status, clearing the status flags, and enabling, disabling and masking interrupts were added following the pattern of existing UC3 peripherals.

The main design of the TMU was kept when it was implemented for the UC3 se-

Offset	Register	Description
0x00	CTRL	Control register
0x04	MODE	Mode register
0x08	SR	Status register
0x0c	SCR	Status clear register
0x10	IER	Interrupt enable register
0x14	IDR	Interrupt disable register
0x18	IMR	Interrupt mask register
0x1c	COMPARE_HI	Compare register
0x20	COMPARE_LO	
0x24	COUNT_HI	Count register
0x28	COUNT_LO	
0x2c	SWAP_COMPARE_HI	Swap compare register
0x30	SWAP_COMPARE_LO	
0x34	SWAP_COUNT_HI	Swap count register
0x38	SWAP_COUNT_LO	

Table 3.1: *User interface of the TMU on the Atmel AVR32 UC3.*

ries. The most noticeable change is that the TMU has been placed on the peripheral bus instead of the high-speed bus as originally intended. However, the UC3 allows the creation of a special CPU local bus to the TMU [53]. If implemented this would provide single-cycle deterministic access to the TMU registers. Another aspect of moving the TMU is that it will be deactivated in some of the system sleep modes that turn off the clock for peripherals. This is not a problem for execution time measurement as only the non-visible execution time clock for idling is active when within sleep modes.

3.7 Ada 2012 execution time control using the TMU

Article A.7 is titled *"Improving the performance of execution time control by using a hardware Time Management Unit"* [25], describes how the TMU was used to reduce the overhead of execution time control, and gives test results obtained from simulation with the synthesizable RTL code of the AVR32 UC3L microcontroller. Only minor changes were needed to the earlier implementation of execution time

control in order to use the TMU.

3.7.1 Implementation with the TMU

The implementation of Ada 2012 execution time control described in Article A.6 was modified to take advantage of the TMU. The original implementation is referred to as CC-ETC, where CC stands for COUNT / COMPARE, while the implementation using the TMU is referred to as TMU-ETC.

The specification of System.BB.Time was not altered when the TMU was used, and there is still one clock and alarm abstraction used both for the RTC and execution time clocks. However, the execution time clocks now use the TMU instead of sharing hardware timer with the RTC. This means that the low-level procedures interfacing with the hardware had to be updated to use different timers. Also the COMPARE interrupt handler had to be updated since it now only serves the RTC, and a new interrupt handler for the TMU was added. In addition an interface to the TMU peripheral unit was added to the package System.BB.Peripherals.

The context switch routine now changes the execution time clock directly as shown in Listing 3.4, not through a wrapper Ada routine done with CC-ETC. The ETC is changed to the Active_Clock of the first thread by first loading Base_Time and First_Alarm.Timeout for this clock. When these values are loaded, the TMU swap operation is initiated using the multiple store instruction of the AVR32 architecture. Notice that the registers are stored in reverse order and therefore the high-word is stored before the low-word. Hence, all swap registers are written before the swap operation is triggered. After the swap operation the COUNT value is read back from the swap register and stored as the Base_Time of the previous ETC. Finally, the ETC is updated to point to the new active execution time clock.

Compared to CC-ETC, the TMU-ETC package body of System.BB.Time has three more statements and five less declarations, meaning that is has two logical code line less than the earlier implementation. For the peripheral packages the differences in code lines also include the changes going from the UC3A to the UC3L microcontroller series. In these packages 50 logical code lines were added for using the TMU, whereof only 8 are statements. For the context switch only 8 additional instructions were needed for using the TMU as seen in Listing 3.4. All of these are simple load, store or move instructions, there are no calculations or branches that complicate the assembler code. In essence the complexity of the run-time environment as a whole is unchanged by using the TMU.

Listing 3.4: *Changing clocks in context switch routine with the TMU.*

```
/* Load Active_Clock of first_thread (stored in r9) */
ld.w    r0, r9[THREAD_ACTIVE_CLOCK_OFFSET]

/* Load First_Alarm.Timeout and Base_Time */
ld.w    r1, r0[CLOCK_FIRST_ALARM_OFFSET]
ld.d    r4, r1[ALARM_TIMEOUT_OFFSET]
ld.d    r2, r0[CLOCK_BASE_TIME_OFFSET]

/* Do TMU swap operation */
mov     r1, TMU_ADDRESS + TMU_SWAP_OFFSET
stm     r1, r2-r5
ld.d    r4, r1[8]

/* Load ETC address */
lda.w   r1, system__bb__time__etc

/* Load current ETC and store its Base_Time */
ld.w    r2, r1
st.d    r2[CLOCK_BASE_TIME_OFFSET], r4

/* Active_Clock of first_thread is now ETC */
st.w    r1, r0
```

3.7.2 Performance improvements

Performance testing is done by simulation as the TMU has not yet been included in an UC3 microcontroller chip. However, the synthesizable RTL code of the UC3 system is used, and the results are therefore as accurate as if the testing was done on real hardware. The run-time environment and test programs are compiled and linked to an ELF file as normal, and no special code or libraries were needed to execute on the simulator. The tests programs are the same with the exception of the non-simulated tests sending data over the USART line, whereas the simulated TMU tests store data in memory because it can be read directly using the simulator. Some updates to the run-time environment were also needed since the simulated microcontroller is of version UC3L, while the earlier tests were for the UC3A [31]. These differences do not affect the test results.

Testing showed that using the TMU for the implementation of Ada 2012 execution time control reduced the overhead and therefore improved the performance

Test	Improvement	
	CPU cycles	Reduction (%)
Context switch overhead	65	54
Interrupt handler overhead	30	25
Timing event overhead	4	4
Interruption cost	42	21

Table 3.2: *Performance improvements with TMU.*

of the system. As seen from the overview in Table 3.2 some improvements were more significant than others. The overhead of handling timing events is hardly reduced at all. This is explained by the RTC now being reset in Compare_Handler before calling the handler in addition to changing to the interrupt clock. The CC-ETC implementation does both in one operation when the ETC is updated and is therefore almost as efficient as TMU-ETC.

For general interrupt handling there is a noticeable reduction of latency caused by the reduced overhead of changing execution time clocks when using the TMU swap operation. Related to this is the improvement in cost to the interrupted task that also has a noticeable improvement. However, the absolute cost of interruption is still quite high.

The best improvement happens for the context switch. This was expected, since the simplicity of the TMU allowed a call to an Ada procedure changing the task clock to be replaced with a few lines of assembly code as seen in Listing 3.4. Combined with the general speed-up of changing clocks for the TMU, this more than halved the overhead introduced by execution time control compared to the earlier implementation. In systems with frequent context switches, this should give a noticeable performance improvement.

Chapter 4

Conclusions and future work

4.1 Evaluation of contributions

The research goals of the PhD project described in this thesis has been successfully accomplished. The main contributions can be separated into three parts:

1. The porting of GNAT to the AVR32 architecture and a bare-board Ravenscar run-time environment to the UC3 microcontroller series.
2. The development of execution time control for interrupt handling and the establishment of usage patterns for this brand new feature. Execution time measurement for interrupts is now in the ISO standard for Ada 2012.
3. The design of a Time Management Unit (TMU) to reduce the overhead of execution time control that has been implemented and tested with the UC3.

Each of these contributions are evaluated in the following.

4.1.1 Ada development for the AVR32 architecture

As a result of the contributions described in this thesis, Ada is now available for the AVR32 architecture for the first time. The high-quality bare-board run-time environment for the UC3 microcontroller series supports the restricted Ravenscar tasking profile with additional support for full execution time control both for tasks and interrupt handling. The object-oriented Ada real-time framework has been adapted to the limitations of the Ravenscar profile and is also available with the run-time environment.

Using the run-time environment and the real-time framework, developing advanced multi-tasking applications for the UC3 microcontroller series is possible for anyone with a basic knowledge of the Ada programming language. This could open for new applications for the AVR32 within high-integrity embedded real-time systems, and also open up new markets for Ada and GNAT. The system is also suitable for educational purposes.

The run-time environment is freely available under the GNU Public License, and can be modified and used by anyone for any purpose as long as changes are shared back to the community. There has already been some academic and commercial interest for the run-time environment.

4.1.2 Execution time control for interrupt handling

Prior to this research work, the execution time of interrupt handling was charged to the interrupted task by all implementations known to the candidate. This caused inaccuracy to the execution time measurement of tasks. By implementing separate execution time measurement for interrupt handling this inaccuracy was reduced to a small constant cost to the interrupted task. This allows task budgets to be tighter, and thereby making higher CPU utilization possible.

However, the contributions of this thesis go further, implementing full execution time control for interrupt handling similarly to that of tasks. This allows accurate and efficient control of the execution time spent handling interrupts for the first time, allowing the system to be protected against unexpected bursts of interrupts caused by hardware errors, design faults or usage errors. This new feature was demonstrated by extending the real-time framework with an interface for interrupt execution time servers and implementing a deferrable server, allowing interrupt handling to be modeled as a periodic task in the scheduling analysis.

Initially, execution time control for interrupt handling was developed for interrupt levels, measuring the execution time of a pseudo task handling all interrupts of a given level. This approach was chosen because it required less memory resources and also made hardware implementation of execution time control easier since the interrupt level is known by the hardware upon entering the low-level handler. When this new feature was presented at IRTAW-14 and proposed added to the next revision of Ada, the alternative proposal of using interrupt IDs instead of levels was favored. The reason for this is primarily that it gives finer control over the execution time of individual interrupts and is also better suited for multiprocessors. While full execution time control for interrupt handling was postponed, execution time measurement for individual interrupt IDs is now included in the ISO standard

draft for the forthcoming Ada 2012.

4.1.3 The hardware Time Management Unit

The final contribution of this thesis is the design of a Time Management Unit (TMU) to reduce the overhead incurred by execution time control. The TMU was designed as a simple memory-mapped 64-bit timer with a special swap operation for efficient atomic change of the running clock. The simple design makes the TMU straightforward to implement for different computer architectures. Under the guidance of Amund Skavhaug and the candidate the TMU has been successfully implemented for the Atmel AVR32 UC3 microcontroller series as part of a master's thesis at NTNU in close cooperation with Atmel Norway.

The implementation of Ada 2012 execution time control has been modified to use this TMU and has been tested on a simulated UC3L microcontroller using Atmels proprietary synthesizable RTL code and tools. Test results have shown that the TMU significantly reduces the overhead of execution time control, both for context switches and interrupt handling. This is important for the adaptation of execution time control, as a too high overhead may not be acceptable for many real-time systems.

4.2 Future work

4.2.1 Execution time control with speed scaling

As of now execution time measurement in Ada is related to the passage of time and not the use of CPU cycles. Therefore a task with a budget of a time span T will be allowed to execute for this amount of time regardless of which speed the CPU is running at. However, the execution time needed by the task to finish its work will clearly be dependent on this speed. It should be clarified how speed scaling is to be handled with execution time control. Also, an API for dynamic speed scaling should be added to Ada in order to support this energy saving feature.

4.2.2 Further development of GNATforAVR32

Further work is needed for GNATforAVR32, particularly to make the system easier to configure and build for different versions of the UC3. The system has already

been updated to an extended version of GNARL, supporting exception propagation, synchronized interfaces and larger parts of the standard library. This will be released to the general public as soon as more testing has been performed.

4.2.3 Further development of the TMU

Experience gained by implementing execution time control with the TMU and evaluation of the performance test results gave insight into some possible improvements to the design. The primary concern is that the interrupt overhead is still high. In order to achieve further reduction to this overhead a hybrid solution between the earlier and current TMU design may be needed to automatically change to interrupt clocks upon entering an interrupt level.

Another possible improvement to the TMU is to use the CPU local bus for accessing the swap registers. This would allow more efficient clock changes with constant execution time.

4.2.4 Thermal aware scheduling

The power wall [38] and thermal gradients [51] are another future challenge for real-time systems. While future architectures may have a vast number of cores running at high frequencies, the power limit is expected to remain constant at 198 W due to packaging constraints with regards to temperature. It is not feasible to dissipate more power while keeping below the temperature threshold. Exceeding this threshold may cause errors in computation, reduce the lifetime of the chip or even cause permanent damage [51]. Temperature gradients may also cause damage to the chip even if the temperature is below the threshold. Power-saving techniques are not enough to counter these problems, motivating modeling and control of the temperature at the architectural level [51].

Several methods for controlling the temperature of computer architectures exist. These may be hardware or software-based, and either reactive or predictive. Many designs already use hardware-based Dynamic Thermal Management (DTM) to avoid dangerous overheating by for instance applying Dynamic Voltage Scaling (DVS) to keep components below the thermal threshold, clock gating to turn off overheating components, and possibly also enabling spare components to reduce the performance penalty [51]. There are also software thermal scheduling techniques that use knowledge of the application load to balance the temperature and keep below the thermal threshold, thereby avoiding the performance loss inflicted when DTM is activated. Examples of such techniques are task migration [39],

speed scaling of processor cores [6], and alternating between running tasks with higher and lower thermal profiles [34]. A novel approach taken by [63] is to use a model of the thermal properties of the system and control theory to assign optimal core speeds to balance the core temperatures.

Amund Skavhaug and the candidate have done some research on simulating the thermal properties at the architecture level of multi-processor system-on-chip (MP-SoC) architectures [27] and controlling the temperature of such systems using model predictive control (MPC) and a genetic algorithm [23]. However, this research has not been included in this thesis as it is not within the main topic.

Bibliography

[1] Ada Rapporeur Group (ARG). Ada issue 307: execution-time clocks. *Ada Lett.*, XXVI:31–44, April 2006.

[2] Atmel Corporation. *AVR32 AP - Technical Reference Manual*, June 2006.

[3] Atmel Corporation. *AVR32 - Architecture Document*, November 2007.

[4] Atmel Corporation. *AVR32UC3 - Technical Reference Manual*, March 2010.

[5] T. Baker and A. Shaw. The cyclic executive model and Ada. In *Proc. Real-Time Systems Symposium*, pages 120–129, 6–8 Dec. 1988.

[6] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1):1–39, 2007.

[7] J. Barnes. *Programming in Ada 2005*. Addison-Wesley, 2006.

[8] J. Barnes. *Rationale for Ada 2005*. John Barnes Informatics, 2007.

[9] J. Barnes. *A brief introduction to Ada 2012*. John Barnes Informatics, 2011.

[10] A. Burns. The Ravenscar profile. *Ada Lett.*, XIX(4):49–52, 1999.

[11] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2):1–74, 2004.

[12] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Pearson, 3 edition, 2001.

[13] A. Burns and A. Wellings. Supporting deadlines and EDF shceduling in Ada. In *Ada-Europe 2004*, pages 156–165. Springer Berlin / Heidelberg, 2004.

[14] A. Burns and A. Wellings. Programming execution-time servers in Ada 2005. In *Proc. 27th IEEE International Real-Time Systems Symposium RTSS '06*, pages 47–56, Dec. 2006.

[15] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge, 2007.

[16] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, H. Falk, and P. Marwedel. A unified WCET analysis framework for multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, Beijing, China, April 2012. (accepted for publication).

[17] J. A. de la Puente and J. Zamorano. Execution-time clocks and Ravenscar kernels. *Ada Lett.*, XXIII(4):82–86, 2003.

[18] J. A. de la Puente, J. Zamorano, J. Ruiz, R. Fernández, and R. García. The design and implementation of the Open Ravenscar Kernel. In *IRTAW '00: Proceedings of the 10th international workshop on Real-time Ada*, pages 85–90, New York, NY, USA, 2001. ACM.

[19] B. Forsman. A Time Management Unit (TMU) for real-time systems. Master's thesis, Norwegian University of Science and Technology (NTNU), 2008.

[20] M. González Harbour. Real-time POSIX: An overview, 1993.

[21] M. González Harbour et al. Implementing and using execution time clocks in Ada hard real-time applications. In *Lecture Notes in Computer Science*, volume Volume 1411/1998, pages 90–101. Springer Berlin / Heidelberg, 1998.

[22] M. González Harbour and M. A. Rivas. Managing multiple execution-time timers from a single task. *Ada Lett.*, XXIII(4):28–31, 2003.

[23] K. N. Gregertsen. Thermal control of MPSoC architectures under a Markovian traffic load using model predictive control and a genetic algorithm.

[24] K. N. Gregertsen and A. Skavhaug. Functional specification for a Time Management Unit. Presented at SAFECOMP 2010.

[25] K. N. Gregertsen and A. Skavhaug. Improving the performance of execution time control by using a hardware Time Management Unit. Accepted for Ada-Europe 2012.

[26] K. N. Gregertsen and A. Skavhaug. An efficient and deterministic multitasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, pages 1572–1575, April 2009.

[27] K. N. Gregertsen and A. Skavhaug. Modeling of thermal properties for prospected future MPSoCs. In *Proceedings of the work in progress session – DSD 2009*, 2009.

[28] K. N. Gregertsen and A. Skavhaug. A real-time framework for Ada 2005 and the Ravenscar profile. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 515–522, Aug. 2009.

[29] K. N. Gregertsen and A. Skavhaug. Execution-time control for interrupt handling. *Ada Lett.*, 30, 2010.

[30] K. N. Gregertsen and A. Skavhaug. Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture. *Journal of Systems Architecture*, 56:509–522, 2010.

[31] K. N. Gregertsen and A. Skavhaug. Implementation and usage of the new Ada 2012 execution time control features. *Ada User Journal*, 32(4):265–275, December 2011.

[32] ISO/IEC. *Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1.*

[33] ISO/IEC. *Ada Reference Manual - ISO/IEC 8652:201x(E) (Draft 15).*

[34] R. Jayaseelan and T. Mitra. Temperature aware scheduling for embedded processors. In *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*, pages 541–546, Washington, DC, USA, 2009. IEEE Computer Society.

[35] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, Porto / Portugal, jul 2011.

[36] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill International Edition, 1997.

[37] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[38] C. Meenderinck and B. Juurlink. (When) will CMPs hit the power wall? In *Euro-Par 2008 Workshops - Parallel Processing*, pages 184–193. Springer-Verlag, Berlin, Heidelberg, 2009.

[39] P. Michaud et al. A study of thread migration in temperature-constrained multicores. *ACM Trans. Archit. Code Optim.*, 4(2):9, 2007.

[40] S. Michell and J. Real. Conclusions of the 14th International Real-Time Ada Workshop. *Ada Lett.*, 30, 2010.

[41] J. Miranda and M. González Harbour. A proposal to integrate the POSIX execution-time clocks into Ada 95. In J.-P. Rosen and A. Strohmeier, editors, *Reliable Software Technologies – Ada-Europe 2003*, volume 2655 of *Lecture Notes in Computer Science*, pages 638–638. Springer Berlin / Heidelberg, 2003.

[42] J. Miranda and E. Schonberg. *GNAT: The GNU Ada Compiler*. Web: `https://www2.adacore.com/gap-static/GNAT_Book/html/`, 2004.

[43] J. Real and A. Crespo. Incorporating operating modes to an Ada real-time framework. *Ada Letters*, 30:73–85, May 2010.

[44] M. A. Rivas and M. González Harbour. Extending Ada's real-time systems annex with the POSIX scheduling services. *Ada Lett.*, XXI:20–26, March 2001.

[45] M. A. Rivas and M. González Harbour. MaRTE OS: An Ada kernel for real-time embedded applications. In D. Craeynest and A. Strohmeier, editors, *Reliable Software Technologies – Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 305–316. Springer Berlin / Heidelberg, 2001.

[46] M. A. Rivas and M. González Harbour. Execution time monitoring and interrupt handlers: position statement. *Ada Lett.*, 30, 2010.

[47] J. F. Ruiz. GNAT PRO for on-board misson-critical space applications. *Ada-Europe*, 2005.

[48] J. F. Ruiz, J. A. de la Puente, J. Zamorano, J. González-Barahona, and R. Fernández-Marina. *Open Ravenscar Real-Time Kernel Software Design Document*, 2001.

[49] S. Sáez, S. Terrasa, and A. Crespo. A real-time framework for multiprocessor platforms using Ada 2012. In *Proceedings of the 16th Ada-Europe international conference on Reliable software technologies*, Ada-Europe'11, pages 46–60, Berlin, Heidelberg, 2011. Springer-Verlag.

[50] O. M. Santos and A. Wellings. Cost enforcement in the real-time specification for Java. *Real-Time Systems*, 37(2):139–179, 2007.

[51] K. Skadron et al. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[52] H. Skinnemoen and A. Skavhaug. Hardware support for on-line execution time limiting of tasks in a low-power environment. In *EUROMICRO / DSD Work in progress session*. Linz: Institute of system science, Johannes Kepler University, 2003.

[53] S. J. Søvik. Hardware implementation of a Time Management Unit. Master's thesis, NTNU, 2010.

[54] The Free Software Foundation (FSF). The GNU Public License (GPL) version 3. Web: `http://www.gnu.org/copyleft/gpl.html`, 2007.

[55] The Open Group. The Open Group base specifications issue 6, IEEE Std 1003.1. Web: `http://www.opengroup.org/onlinepubs/000095399/`.

[56] S. Urueña, J. Pulido, J. Redondo, and J. Zamorano. Implementing the new Ada 2005 real-time features on a bare board kernel. *Ada Lett.*, XXVII(2):61–66, 2007.

[57] J. Uthus and Ø. Strøm. MCU architectures for computer-intensive embedded applications. Technical report, Atmel Corporation, 2005.

[58] T. Vardanega, M. G. Harbour, and L. M. Pinho. Session summary: language and distribution issues. *Ada Lett.*, 30, 2010.

[59] A. Wellings. Implementation experience with Ada 2005. *Ada Lett.*, XXVII, no 2:59–60, 2007. session report.

[60] A. Wellings and A. Burns. *Ada-Europe 2007*, chapter Real-Time Utilities for Ada 2005, pages 1–14. Springer Berlin / Heidelberg, 2007.

[61] A. J. Wellings and A. Burns. A framework for real-time utilities for Ada 2005. *Ada Lett.*, XXVII(2):41–47, 2007.

[62] R. Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[63] F. Zanini, D. Atienza, and G. De Micheli. A control theory approach for thermal balancing of MPSoCs. *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 37–42, Jan. 2009.

Appendix A

Published material

In this appendix the published material forming the basis of this thesis is included. The papers are included as they appear in the publications, with the exception of the header and footer being removed.

A.1 Article No. 1

K. N. Gregertsen and A. Skavhaug:

"An efficient and deterministic multi-tasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller"

Published in the proceedings of DATE'09 [26].

An efficient and deterministic multi-tasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR®32 UC3 microcontroller

Kristoffer Nyborg Gregertsen
Dept. of Engineering Cybernetics
NTNU, Norway
gregerts@itk.ntnu.no

Amund Skavhaug
Dept. of Engineering Cybernetics
NTNU, Norway
amund@itk.ntnu.no

Abstract

This paper describes how an efficient and deterministic multitasking run-time environment supporting the Ravenscar tasking model of Ada 2005 was implemented on the Atmel AVR32 UC3A microcontroller. The open source GNU Ada Compiler (GNAT GPL 2007) was also ported to AVR32 as a part of this work, making a working Ada development environment available on the architecture for the first time.

1. Introduction

Dependability is essential in many embedded and real-time systems. Failures can often result in financial losses, environmental damage and even the loss of lives. Ada is a programming language designed for high-integrity systems and has many safeguards against common programming errors. The language is an ISO standard [1], making programs portable within different compilers, run-time libraries and operating systems.

While Ada is much used within high-integrity systems, the concurrent constructs of the language have often been excluded as being non-deterministic and inefficient [2]. Instead such methods as the cyclic executive [3], [4] has been used. Advances in static analysis have made it possible to check hard deadlines when using preemptive fixed priority scheduling. This has led to development of the Ravenscar profile [5], a subset of the Ada tasking model designed to provide the static and deterministic environment needed to perform static analysis [2]. The simplicity of the tasking model also allows efficient run-time environments.

The Polytechnical University of Madrid developed the Open Ravenscar Kernel (ORK) [6] on contract to the European Space Agency for the ERC32 architecture. The Open Ravenscar Kernel was further developed and integrated into the GNU Ada Run-Time Library (GNARL) by José F. Ruiz at AdaCore [7].

The Atmel AVR32 [8] is a brand new architecture designed by Atmel Norway in cooperation with the Norwegian University of Science and Technology (NTNU), and is optimized for code density and high computational throughput with low power consumption [9]. By porting the GNU Ada Compiler (GNAT) to AVR32 and the bare-board Ravenscar run-time environment to the UC3A microcontroller [10] Ada is made available on this architecture for the first time.

It is shown how the simplicity and power of the AVR32 architecture and the UC3A microcontroller combined with the restricted Ravenscar tasking model allows the multi-tasking run-time environment to be deterministic and efficient, making it well suited for high-integrity embedded applications.

2. The Ravenscar profile

The Ravenscar profile is specified as a set of configuration pragmas [1], [5] defining restrictions to the Ada tasking model and the required dynamic semantics. The following features are supported [2]:

- Tasks types and objects defined at library level.
- Protected types and objects, defined at library level, limited to one entry having a simple guard and a queue length of one.
- Ceiling Locking policy with FIFO dispatching policy within priorities.
- The Ada.Real_Time package for high-precision timing and the **delay until** statement.
- Synchronous task control, including suspension objects for simple synchronization.
- Protected procedures as statically bounded interrupt handlers.

The sequential parts of Ada are not affected by the profile [2].

3. GNARL

The Ravenscar version of the GNU Ada Run-Time Library (GNARL) is designed to take advantage of the simplifications allowed by the profile [7]. Task management is simplified since all tasks are at library level, cannot terminate and have fixed priority. All task data structures are statically allocated, thus memory requirements are determined at link time. Protected objects are simplified since there are no asynchronous operations, no time-out on entry calls and no varying queue length on entries. Evaluation on protected entries may be done by proxy, thereby improving performance by reducing the number of context switches [7].

The GNU Low-Level Library (GNULL) is a translation layer between generic and actual operating system calls on most systems, but in this case it fully implements the needed dynamic semantics of the Ravenscar profile by including a multitasking core based on the Open Ravenscar Kernel [6], [7].

The core implements preemptive fixed priority scheduling with ceiling locking, having 256 priorities including the interrupt priorities. The number of priorities can easily be changed if needed. Each interrupt priority has its own interrupt stack allowing interrupt nesting while avoiding priority inversion [7]. Interrupts are masked as long as there is a task with higher or equal priority to that interrupt and all interrupts are masked while modifying core data.

The timing services of the core provides as high precision as possible while supporting the needed 50 year time span. This is done by using a 64-bit value for time divided into two parts. The least significant part is present in the hardware timer, while the most significant part is stored in memory and is incremented every time the hardware timer overflows.

4. The AVR32 architecture

The Atmel AVR32 architecture [8] is a 32-bit RISC architecture designed for high computational throughput with low power consumption [9]. The architecture defines instruction lengths of both 16 and 32-bits for high code density and there is a rich set of load / store instructions for high efficiency, supporting byte, half-word, word and double word memory access. The register file of the AVR32 architecture is fairly small having only 13 general purpose registers (R0 to R12), the link register (LR) used for storing routine return addresses, the program counter (PC) and the system register (SR).

The UC3 core [10] is the second implementation of the AVR32 architecture and is primarily intended for embedded control applications where deterministic execution times is important. The UC3 has an internal SRAM integrated with the CPU pipeline in order to bypass the system bus. This allows deterministic, single-cycle read/write memory access. The UC3 fully implements the DSP instructions of the AVR32 ISA such as single-cycle multiply and accumulate instructions for both modular and saturated arithmetic. Atmel claims it to deliver 1.3 Dhrystone MIPS / MHz.

5. Porting to the AVR32 architecture

5.1. Hardware setup

The EVK1100 evaluation board with the UC3A0512 microcontroller [10] was used for developing and testing the run-time environment. The UC3A0512 has 64 KB of internal SRAM, 512 KB of internal flash and is clocked by a 12 MHz external oscillator. The Atmel JTAG ICE Mk II was used for programming and debugging the device.

5.2. Porting the GNAT front-end

The GNU Ada compiler (GNAT) is an Ada front-end for the GNU Compiler Collection (GCC) developed at the University of New York and is now maintained by AdaCore. The GCC back-end for AVR32 was developed at the Norwegian University of Science and Technology and is now maintained by Atmel Norway. Both the front-end and back-end are open source software licensed under the GNU Public License (GPL).

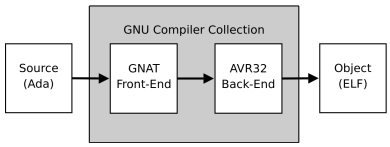

Figure 1. The GNU Compiler Collection.

Since both the front-end and back-end are open source components of GCC, porting GNAT to the AVR32 architecture was much matter of applying the GNAT GPL 2007 patches from AdaCore to the already patched GCC version 4.1.2 source code supplied by Atmel. There were however some incompatibilities caused by register promotion of return values from functions. A quick fix disabling register promotion was provided by Atmel, the problem should however be investigated further.

5.3. Porting the Ravenscar run-time

Only the code that needed to be changed due to differences between the ERC32 and the AVR32 was altered when porting the run-time environment.

5.3.1. Context switch. The context switch code consists of only 15 instructions with no branches:

```
/* Get address of running thread */
lda.w   r8, running_thread
ld.w    r9, r8[0]

/* Save context of running thread */
sub     r9, -48
stm     --r9, r0-r7, sp, lr
mfsr    r0, SYSREG_SR
st.w    --r9, r0
st.w    --r9, r12

/* Get address of first thread */
lda.w   r1, first_thread
ld.w    r9, r1[0]

/* First thread is now running thread */
st.w    r8[0], r9

/* Load context of first thread */
ld.w    r12, r9++
ld.w    r0, r9++
mtsr    SYSREG_SR, r0
sub     pc, -2
ldm     r9++, r0-r7, sp, pc
```

The addresses of the running and first thread are stored in memory instead of being passed as arguments for debugging purposes [7].

5.3.2. Interrupt handling. The AVR32 has a peripheral interrupt controller which groups different interrupt lines. Each interrupt group is assigned to one of the 4 interrupt levels by the software driver.

There is a low-level handler for each interrupt level. The interrupt ID is found by reading the interrupt cause register of the level to find the interrupt group, and then the interrupt request register of that group to find the interrupt line. The highest numbered asserted line is chosen if there are more than one.

The handler loads the interrupt stack for the given level and calls the interrupt wrapper with the interrupt ID. Prior to returning from the handler the task stack pointer is restored and a context switch is done if needed.

5.3.3. Peripheral drivers. The power manager unit is used to enable the external oscillator upon initialization of the system and setup the CPU and peripheral clocks relative to it. For simplicity, it was chosen to run both the CPU and the peripherals at the same clock rate as the external clock.

The interrupt controller provides the functionality to activate interrupts and read the interrupt ID. The package provides the mapping between interrupt identities and groups, and is specific to a given MCU series. The interrupt priorities are also defined in this package.

Two 16-bit counters are used by the timing services. One counter is used as the least significant part of the system clock, counting the whole 16-bit range and generating an interrupt on overflow. The other counter is used in one-shot mode for setting off alarms between the regular interrupts, allowing fine grained task release.

6. Metrics

6.1. Code size

The context switch for the AVR32 consists of only 15 assembler code lines. The number of assembler code lines for interrupt handling is only 18 compared to more than 100 for the ERC32. In total the number of assembler code lines is reduced from about 400 with the ERC32 to about 50 with the AVR32. The AVR32 implementation needs more peripheral drivers resulting in more Ada code lines as seen from Table 1. Most of the added lines are however register declarations for the peripherals and not executable statements.

Table 1. Comparison of Ada code metrics.

Metric	ERC32	AVR32
Statements	261	378
Declarations	528	896
Total	789	1274

6.2. Memory requirements

The memory requirements of the multitasking core are generally low, only the interrupt handling package uses a noticeable amount of SRAM memory due to the interrupt stacks in its BSS section. The size of the text section used by the run-time core is about 5.5 KB which is just above 1% of the total Flash memory available on the UC30512.

6.3. Performance

A simple test of the time needed to switch context was performed by having one task assert an external pin, unblock a second task and then go to sleep, when the second task started executing it negated the same pin. The time the external pin was asserted was measured to be about 15 μs when the system was running on 12 MHz, this equals approximately 180 clock cycles.

7. Discussion

7.1. Choice of hardware

The AVR32 is a brand new architecture with an instruction set created from scratch, making it interesting for research purposes. The architecture was created by Atmel Norway in Trondheim in collaboration with the NTNU located in the same city. The relationship between Atmel Norway and NTNU makes it possible to later design and test new hardware solutions supporting the run-time environment together with the code for the AVR32 core.

The UC3 core was chosen over the more powerful AP7 core out of several reasons. System implementation was easier on the UC3 since it requires less software drivers. However the deterministic one-cycle access time to internal SRAM was the primary reason for choosing the UC3. The AP7 has a higher average performance, but relies on external cached SDRAM resulting in a high worst-case time for memory access.

7.2. Multitasking core

The context switch is highly efficient and has a constant execution time, avoiding the problem of having a worst-case execution time that is significantly longer than the average. This should make it easier to preform accurate static analysis of applications.

The timer/counter module of the UC3 is only 16-bit causing frequent overflows. Division of the clock signal used by the timer reduces the frequency of timing interrupts but also the resolution of the system clock. Which is preferred depends on the application.

The programmable interrupt handling model of the AVR32 is very flexible. Different applications may assign different priorities to interrupt groups instead of having these decided by hardware. Only a modification of constants in the interrupt specification file and a recompilation is needed to change the priorities.

7.3. Memory requirements

The memory requirements are dominated by the task and interrupt stacks. The number of interrupt stacks is reduced from 15 on the ERC32 implementation to only 4 on the AVR32, reducing the amount of memory needed for these stacks with more than 73%. The size of the stacks may easily be altered. The secondary stack is used for returning objects of variable size from routines, and may be removed altogether for some systems further reducing the memory requirements.

8. Conclusion

The simplicity and power of the AVR32 architecture allows several enhancements compared to the original ERC32 implementation with regards to determinism, analyzability and efficiency. In particular the context switch and interrupt handling code are simplified. The context switch execution-time is deterministic easing schedulability analysis.

By porting the GNAT to AVR32 Ada is made available on this architecture for the first time. This could open new applications for AVR32 within high-integrity embedded real-time systems, and also open new markets for Ada 2005 and GNAT. The system may also be suitable for educational purposes.

Acknowledgment

Many thanks to José F. Ruiz and Arnauld Charlet at AdaCore for guidance on porting GNAT. Thanks to Atmel Norway for providing hardware and tools, and Ronny Pedersen for supporting the GCC back-end.

References

[1] ISO/IEC, *Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1*. [Online]. Available: http://www.adaic.com/standards/05rm/html/RM-TOC.html

[2] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar profile in high integrity systems," *Ada Lett.*, vol. XXIV, no. 2, pp. 1–74, 2004.

[3] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 3rd ed. Pearson, 2001.

[4] T. Baker and A. Shaw, "The cyclic executive model and Ada," in *Proc. Real-Time Systems Symposium*, 6–8 Dec. 1988, pp. 120–129.

[5] A. Burns, "The Ravenscar profile," *Ada Lett.*, vol. XIX, no. 4, pp. 49–52, 1999.

[6] J. A. de la Puente, J. Zamorano, J. Ruiz, R. Fernández, and R. García, "The design and implementation of the Open Ravenscar Kernel," in *IRTAW '00: Proceedings of the 10th international workshop on Real-time Ada*. New York, NY, USA: ACM, 2001, pp. 85–90.

[7] J. F. Ruiz, "GNAT pro for on-board misson-critical space applications," *Ada-Europe*, 2005.

[8] Atmel Corporation, *AVR32 - Architecture Document*.

[9] J. Uthus and Ø. Strøm, "MCU architectures for computer-intensive embedded applications," Atmel Corporation, Tech. Rep., 2005.

[10] Atmel Corporation, *AT32UC3A Series - Preliminary Datasheet*.

A.2 Article No. 2

K. N. Gregertsen and A. Skavhaug:

"A real-time framework for Ada 2005 and the Ravenscar profile"

Published in the proceedings of SEAA 2010 [24].

A real-time framework for Ada 2005 and the Ravenscar profile

Kristoffer Nyborg Gregertsen, Amund Skavhaug
Department of Engineering Cybernetics, NTNU
N-7491 Trondheim, Norway
{*gregerts,amund*}*@itk.ntnu.no*

Abstract

This paper describes an object-oriented real-time framework for Ada 2005 and the Ravenscar profile. The framework uses the Ada 2005 real-time features implemented on the AVR32 UC3 microcontroller series to control the execution-time of tasks and interrupt handlers. An example application using the framework and test results are given.

1. Introduction

Dependability is essential for embedded and real-time systems as failure of such systems may result in financial losses, environmental damage or even the loss of lives. As the complexity of embedded software increases system failures caused by software faults is getting ever more prevalent.

Failures are often classified as being either in the value or time domain. The Ada programming language [1] was designed for use in high-integrity systems and has many safeguards to prevent failures in the value domain. The language is strongly typed and has strict checks for access types, making many common programming faults being detected already at compile time. Furthermore the language has run-time checks to catch errors not detectable at compilation time.

Failures in the time domain may be harder to prevent as the errors may occur due to obscure faults such as race conditions in the underlying tasking environment. The Ada tasking model with its rich set of synchronization primitives has traditionally not been considered suitable for high-integrity systems due to its non-deterministic nature. The Ravenscar profile [2] standardized as a part of Ada 2005 defines a sub-set of the tasking model designed to provide the static and deterministic environment needed for such systems.

Static analysis techniques for the time domain such as Rate Monotonic Analysis (RMA) [3] relay on tasks not exceeding their execution-time budget. These budgets may be determined by analysis on the task code in order to estimate its Worst-Case Execution-Time (WCET). However this analysis may not be trivial on simple architectures and may be very hard and time consuming for more advanced architectures with features such as deep pipelines, cache and branch prediction [4]. This leads to the use of conservative estimates for the budgets. Furthermore the WCET will often be much greater than the average execution-time.

An alternative is to use less conservative budgets and handle execution-time overruns dynamically. In order to facilitate this Ada 2005 defines execution-time clocks and timers [1]. The Ravenscar profile does not allow execution-time timers. However, research prior to the writing of the Ada 2005 standard concluded that these timers were useful and compatible with the Ravenscar profile [5].

A framework for common real-time paradigms such as periodic and sporadic tasks with detection of deadline misses and execution-time overruns may be useful in order to ease development and reduce the number of software faults. Such a real-time framework was developed for Ada 2005 by Andy Welling and Alan Burns [6]. Their framework is however not compatible with the tasking model of the Ravenscar profile. This paper seeks to implement a version of the framework compliant with the Ravenscar profile with the exception of using execution-time timers.

The Atmel AVR32 [7] is a brand new architecture designed by Atmel Norway in cooperation with the Norwegian University of Science and Technology (NTNU). The UC3 microcontroller series [8] is the second implementation of the architecture. The GNU Ada Compiler (GNAT) and the GNAT bare-board run-time environment [9] were ported to UC3 at NTNU, and are used for demonstrating the framework.

In the following there is an introduction to the Ada 2005 real-time features and their implementation on the AVR32 GNAT bare-board run-time environment [10], [11]. Then follows a description of the developed

real-time framework. An example application using the framework and test results are given.

2. The Ravenscar profile

The Ravenscar profile is specified as a set of pragmas [1], [2], [12] defining restrictions to the Ada tasking model and the required dynamic semantics.

A static set of tasks and protected objects is achieved by only allowing such objects to be statically declared at library level and disallowing task termination. Dynamic attachment of interrupt handlers and dynamic change of task priorities (with the exception of changes caused by ceiling locking) is also prohibited. Tasks may not have entries, thereby only allowing task communication and synchronization through protected objects or suspension objects.

Several restrictions are applied to ensure a deterministic execution model. A protected object may have a single entry with a queue length of one using a simple barrier. The requeue statement and asynchronous control are disallowed. Relative delay statements are prohibited and the profile forces the use of the real-time package for timing purposes.

The profile requires that the task dispatching model to be used is `FIFO_Within_Priorities` and that `Ceiling_Locking` should be used for protected objects.

3. The Ada 2005 real-time features

Timing events allow protected procedures to be called at a specified time without the need for a task or delay statement [1, D.15]. The Ravenscar profile allows timing events declared at library level. The tagged type `Timing_Event` is *set* with a handler of the type `Timing_Event_Handler` that identifies a protected procedure to be executed when the timing event *occurs*. A timing event may be set to occur at a absolute time or a relative time.

The package `Ada.Execution_Time` defines the type `CPU_Time` for representing the execution-time of a task, which is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf [1, D.14]. The language standard does not specify which task, if any, that is to be charged the execution-time of interrupt handlers. The AVR32 implementation does not charge the interrupted task the execution-time of interrupt handlers [11]. This is instead charged a pseudo interrupt task for the given interrupt priority. Also a task executing an entry by proxy is not charged the execution-time of the entry, this time is charged the task blocked on the entry.

The package `Ada.Execution_Time.Timers` is used for execution-time events [1, D.14.1]. The tagged type `Timer` represents an execution-time event for a single task and is capable of detecting execution-time overruns. The type `Timer_Handler` identifies a protected procedure to be executed when the timer *expires*. A timer is set and cancelled the same way as timing events with `CPU_Time` as absolute time instead of `Time`. The AVR32 implementation defines the type `Interrupt_Timer`, which extends `Timer` and takes the interrupt priority as discriminant [11]. Each interrupt priority but the highest one has an interrupt timer.

4. The real-time framework

The framework presented below is based on a framework by Andy Wellings and Alan Burns [6] referred to as *the original framework*. It has some extensions to the original, supporting more flexible interrupt handling and interrupt execution-time servers. Execution-time servers for tasks are not implemented due to the limitations of the Ravenscar profile.

4.1. Task states

The task state contains the code to be executed by tasks, the associated state and procedures to be executed in case of deadline misses and execution-time overruns:

```
package Task_States is
   type Task_State is abstract tagged limited
      record
         Tid : aliased Task_Id := Null_Task_Id;
         Budget : Time_Span := Time_Span_Last;
         Recovery : Time_Span := Time_Span_Last;
      end record;
   procedure Initialize
     (S : in out Task_State) is abstract;
   procedure Code
     (S : in out Task_State) is abstract;
   procedure Deadline_Miss
     (S : in out Task_State) is null;
   procedure Overrun
     (S : in out Task_State) is null;
   type Any_Task_State is
     access all Task_State'Class;
end Task_States;
```

The task state differs from that of the original framework by not including an explicit deadline, the deadline is instead implicitly defined by properties of the inheriting task states, and by adding the task ID of the underlying task and the recovery time, which is the time a task has to voluntarily stop executing after an overrun.

Two types inheriting `Task_State` are defined in child packages as in the original framework: the `Periodic_Task_State` which adds `Period` and the `Sporadic_Task_State` which adds `MIT` (minimal-interrelease time).

4.2. Release mechanisms

The framework defines sporadic and periodic release mechanisms with and without overrun detection in child packages. Only the release mechanisms with overrun detection are detailed. The interface for release mechanisms is defined as:

```
package Release_Mechanisms is
   type Release_Mechanism is limited interface;
   procedure Wait_For_Next_Release
     (R : in out Release_Mechanism) is abstract;
   type Any_Release_Mechanism is
     access all Release_Mechanism'Class;
   type Open_Release_Mechanism is
      limited interface and Release_Mechanism;
   procedure Release
     (R : in out Open_Release_Mechanism) is abstract;
   type Any_Open_Release_Mechanism is
     access all Open_Release_Mechanism'Class;
end Release_Mechanisms;
```

The definition of release mechanisms is the same as in the original framework with exception of being defined as limited interfaces instead of synchronized interfaces due to the run-time environment not supporting the latter, and the addition of an interface for release mechanisms with a public release procedure.

4.2.1. Periodic. `Controlled_Periodic_Release` implements `Release_Mechanism` and includes a protected object of the private type `Mechanism` implementing the actual release mechanism:

```
protected type Mechanism
  (S : Any_Periodic_Task_State) is
   procedure Initialize;
   entry Wait;
   pragma Priority (System.Any_Priority'Last);
private
   procedure Release (TE : in out Timing_Event);
   procedure Overran (TM : in out Timer);
   Event_Period : Timing_Event;
   Execution_Timer : access Timer;
   Next : Time;
   Open : Boolean := False;
end Mechanism;
```

The timing event and handler `Release` are used to release the task blocked on the entry `Wait` periodically and also set the budget for the task:

```
procedure Release (TE : in out Timing_Event) is
begin
   if Wait'Count = 0 then
      S.Deadline_Miss;
   end if;
   Execution_Timer.Set_Handler
     (S.Budget, Overran'Access);
   Open := True;
   Next := Next + S.Period;
   TE.Set_Handler (Next, Release'Access);
end Release;
```

The first release is on the system epoch time. The deadline of a task is the same as its period. If the task is not blocked on the entry when released it has missed its deadline. The handler `Overran` simply sets itself to be called again after the recovery time has passed and calls the `Overrun` procedure of the task state.

```
procedure Overran (TM : in out Timer) is
begin
   TM.Set_Handler (S.Recovery, Overran'Access);
   S.Overrun;
end Overran;
```

The entry `Wait` has `Open` as guard which is set to false when the entry is executed.

4.2.2. Sporadic. `Controlled_Sporadic_Release` implements `Open_Release_Mechanism` and also includes a private protected type `Mechanism` implementing the actual release mechanism:

```
protected type Mechanism
  (S : Any_Sporadic_Task_State) is
   procedure Initialize;
   procedure Release;
   entry Wait;
   pragma Priority (Any_Priority'Last);
private
   procedure Release_Allowed
     (TE : in out Timing_Event);
   procedure Overran (TM : in out Timer);
   Execution_Timer : access Timer;
   Event_MIT : Timing_Event;
   Released : Boolean := False;
   Allowed : Boolean := False;
   Open : Boolean := False;
end Mechanism;
```

The timing event and the handler `Release_Allow` enforce the minimal-interrelease time (MIT):

```
procedure Release is
begin
   Released := True;
   Open := Allowed;
end Release;

procedure Release_Allowed
  (TE : in out Timing_Event) is
begin
   if Wait'Count = 0 then
      S.Deadline_Miss;
   end if;
   Allowed := True;
   Open := Released;
end Release_Allowed;
```

The deadline of the task is the same as the MIT. The task has missed its deadline if not blocked on `Wait` when the handler is called. The task budget and MIT event are set when the task is released:

```
entry Wait when Open is
begin
   Execution_Timer.Set_Handler
     (S.Budget, Overran'Access);
   Event_MIT.Set_Handler
     (S.MIT, Release_Allowed'Access);
   Released := False;
   Allowed := False;
   Open := False;
end Wait;
```

The procedure `Overran` is similar to the that of the controlled periodic release mechanism.

4.3. Real-time task

The task type `Real_Time_Task` has its priority, an access to a task state and an access to a release mechanisms as discriminants, and the following body:

```
task body Real_Time_Task is
begin
   S.Tid := Current_Task;
   S.Initialize;
   loop
      R.Wait_For_Next_Release;
      S.Code;
   end loop;
end Real_Time_Task;
```

The only difference from the original framework is the statement initializing the task ID.

4.4. Interrupt handling

In order to support more flexible interrupt handling an interface for interrupt states resembling that of task states is added to the framework:

```
package Interrupt_States is
   type Interrupt_State is limited interface;
   procedure Handler
     (S : in out Interrupt_State) is abstract;
   procedure Enable
     (S : in out Interrupt_State) is abstract;
   procedure Disable
     (S : in out Interrupt_State) is abstract;
   type Any_Interrupt_State is
     access all Interrupt_State'Class;
end Interrupt_States;
```

Device drivers for peripherals with interrupts may implement this interface and provide a handler and procedure for enabling and disabling the interrupt. A protected object for interrupt handling resembling the `Real_Time_Task` is defined as:

```
protected type Interrupt_Handler
  (Id  : Interrupt_Id;
   Pri : Interrupt_Priority;
   S   : Any_Interrupt_State) is
   pragma Interrupt_Priority (Pri);
private
   procedure Handler;
   pragma Attach_Handler (Handler, Id);
end Interrupt_Handler;
```

The private handler of the protected objects simply calls the handler of the interrupt state.

4.5. Interrupt servers

Interrupt servers take advantage of the interrupt level timers implemented on the AVR32 architecture to control the execution-time spent on handling interrupts of a given interrupt priority. The interrupt servers are therefore not portable to other run-time environments. An interface for these servers is defined as:

```
package Interrupt_Servers is
   type Interrupt_Server_Parameters is
      record
         Pri : Interrupt_Priority;
         Budget : Time_Span;
         Period : Time_Span;
      end record;
   type Interrupt_Server is limited interface;
   procedure Register
     (S : in out Interrupt_Server;
      I : Any_Interrupt_State) is abstract;
   type Any_Interrupt_Server is
     access all Interrupt_Server;
end Interrupt_Servers;
```

The interrupt server parameters give the interrupt priority, budget and replenishing period of the interrupt server. A tagged type `Deferrable_Interrupt_Server` implements the interface, takes an access to interrupt server parameters as discriminant, and has a private protected object defined as:

```
protected type Mechanism
  (Param : access Interrupt_Server_Parameters)
is
   procedure Register (I : Any_Interrupt_State);
   pragma Priority (Any_Priority'Last);
private
   procedure Replenish (TE : in out Timing_Event);
   procedure Overran (TM : in out Timer);
   Replenish_Event : Timing_Event;
   Execution_Timer : access Interrupt_Timer;
   Next : Time;
   Disabled : Boolean := True;
   Registered : Natural := 0;
   States : State_Array;
end Mechanism;
```

The interrupt execution-time is controlled by using the interrupt timer of the given level and a timing event replenishing the budget. All registered interrupts are disabled on overrun and enabled every time the budget is replenished:

```
procedure Replenish (TE : in out Timing_Event) is
begin
   Execution_Timer.Set_Handler
     (Param.Budget, Overran'Access);
   if Disabled then
      Disabled := False;
      for I in 1 .. Registered loop
         States (I).Enable;
      end loop;
   end if;
   Next := Next + Param.Period;
   TE.Set_Handler (Next, Replenish'Access);
end Replenish;

procedure Overran (TM : in out Timer) is
begin
   if not Disabled then
      Disabled := True;
      for I in 1 .. Registered loop
         States (I).Disable;
      end loop;
   end if;
end Overran;
```

Interrupts are assumed to be initially disabled. The first replenish event, which enables all registered interrupts, is set to occur at the system epoch time when the first interrupt state is registered.

5. Example application

An example application was developed to demonstrate the use of the real-time framework. The application has an interrupt handler for an external interrupt signal, a sporadic task released after a given number of external interrupts and a set of periodic tasks with hard deadlines. In order to ease implementation the tasks busy-wait for a given number of milliseconds instead of executing a "real" algorithm.

The Atmel EVK1100 evaluation board with the AVR32 UC3A0512 microcontroller [8] was used for the application. In addition the AVR Butterfly evaluation board with the AVR ATMega 169 microcontroller [13] was used for generating an external interrupt signal used by the application. Test data was sent on the USART channel from the EVK1100 to the PC serial port where it was retrieved and analyzed. The Atmel JTAG ICE Mk II was used for programming the microcontrollers. A schematic view of the system is shown in Figure 1.

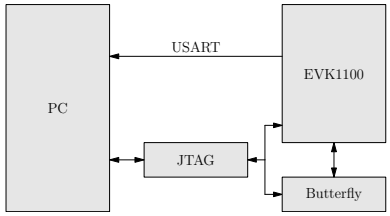

Figure 1. Hardware schematics.

The application has two sources of randomness, the pseudo-random number generator implemented on the AVR32 and the pseudo-random external interrupt signal generated by the ATMega 169 microcontroller.

The "Minimal Standard" generator [14] was implemented for generating a pseudo-random discrete and uniform distribution X in the range 0 to 99. The equation for generating the sample x_i for the internal state s_i is:

$$s_0 = S \tag{1}$$

$$s_{i+1} = 7^5 \cdot s_i \mod (2^{31} - 1) \tag{2}$$

$$x_i = \frac{100 \cdot s_i}{2^{31}} \tag{3}$$

The initial seed s_0 of the generator instances is set to the same non-zero value every time the application is executed.

The external interrupt signal X was generated using the "primitive polynomials modulo 2" method [14]. The equation for sample x_i is:

$$s_0 = 1 \tag{4}$$

$$s_{i+1} = (2 \cdot s_i) \mod 2^{32} + x_i \tag{5}$$

$$x_i = s_{i,18} \oplus s_{i,5} \oplus s_{i,3} \oplus s_{i,1} \tag{6}$$

The signal X has equal probability for high and low values at any sample, thus there is on average one edge every two samples and a falling edge every four samples. The signal was sampled at 800 Hz giving an average time between falling edges of 5 ms.

5.1. Tasks

The tagged type `Simulated_Periodic` extends `Periodic_Task_State` and has the period `T` and budget `C` in microseconds as discriminants. It has a pseudo-random number generator `Gen` and an atomic flag `Timeout` that is used for voluntarily giving up control of the processor. The initialization procedure resets the generator, and sets the budget and recovery time of the task state as follows:

```
procedure Initialize
  (S : in out Simulated_Periodic) is
begin
   Reset (S.Gen, 7 * S.T + 13 * S.C);
   S.Period   := Milliseconds (S.T);
   S.Recovery := Microseconds (250);
   S.Budget   := Milliseconds (S.C) - S.Recovery;
end Initialize;
```

Notice that the actual budget for the task is the budget `C` used in the analysis minus the recovery time which is 250 microseconds. The code executed by the simulated tasks is:

```
procedure Code (S : in out Simulated_Periodic) is
   W : Integer;
begin
   S.Timeout := False;
   if Random (S.Gen'Access) < 50 then
      W := (30 * S.C) / 4;
   else
      W := (50 * S.C) / 4;
   end if;
   for I in 1 .. W loop
      Busy_Wait (100);
      exit when S.Timeout;
   end loop;
end Code;
```

The code needs 75% of `C` to complete half of the times executed and 125% the rest, not including overhead. In the latter case the code will need more execution-time than budgeted. The code is then executed while polling the timeout flag every 100 microseconds. The procedure is exited when the work is done or the timeout flag set.

The timeout flag is set by the `Overrun` procedure. If the procedure is called when the timeout flag is already set an overrun exception is raised. The `Deadline_Miss` procedure raises a deadline miss exception.

The tagged type `Simulated_Sporadic` used for the sporadic task extends `Sporadic_Task_State` and is implemented similarly to `Simulated_Periodic` with the exception of the period being exchanged with the maximal-interrelease time (MIT).

5.2. Interrupt handling

The tagged type `Simulated_Interrupt` extends `External_Interrupt` which again implements `Interrupt_State` and defines the procedure to enable and disable external interrupts. It has the external interrupt `Id`, the number of microseconds for the handler to execute `C`, the number of invocations between releases `N`, and an access to an open release mechanism `R` as discriminants. The handler of the interrupt state takes slightly more than `C` microseconds to execute and calls `R.Release` on every `N` invocation:

```
procedure Handler (S : in out Simulated_Interrupt) is
begin
   S.Clear;
   Busy_Wait (S.C);
   S.Count := S.Count + 1;
   if S.Count mod S.N = 0 then
      S.R.Release;
   end if;
end Handler;
```

While the execution-time of the handler itself is constant there will be an varying overhead associated with handling the interrupt.

5.3. Scheduling

The application task set has an interrupt pseudo task I for the interrupt level of the external interrupt, a sporadic task S, and four periodic tasks A, B, C and D. The scheduling parameters are shown in Table 1.

Table 1. Application task set

Task	T [ms]	C [ms]	U
I	25	2	0.08
S	25	10	0.40
A	25	2	0.08
B	50	5	0.10
C	100	20	0.20
D	200	20	0.10

The total CPU utilization of the tasks set is 0.96 so the utilization-based schedulability test fails, yet as seen from the Gantt chart in Figure 2 all tasks are expected to meet their deadlines.

5.4. Execution

Instances of the simulated periodic and sporadic task states with the parameters as in Table 1 were declared and combined with controlled release mechanism and real-time tasks:

```
S_S : aliased Simulated_Sporadic ( 25, 10);
S_A : aliased Simulated_Periodic ( 25,  2);
S_B : aliased Simulated_Periodic ( 50,  5);
S_C : aliased Simulated_Periodic (100, 20);
S_D : aliased Simulated_Periodic (200, 20);

R_S : aliased Controlled_Sporadic_Release (S_S'Access);
R_A : aliased Controlled_Periodic_Release (S_A'Access);
R_B : aliased Controlled_Periodic_Release (S_B'Access);
R_C : aliased Controlled_Periodic_Release (S_C'Access);
R_D : aliased Controlled_Periodic_Release (S_D'Access);

T_S : Real_Time_Task (200, S_S'Access, R_S'Access);
T_A : Real_Time_Task (190, S_A'Access, R_A'Access);
T_B : Real_Time_Task (180, S_B'Access, R_B'Access);
T_C : Real_Time_Task (170, S_C'Access, R_C'Access);
T_D : Real_Time_Task (160, S_D'Access, R_D'Access);
```

The simulated interrupt state, an interrupt handler and an interrupt server with the parameters from Table 1 were declared as:

```
S_I : aliased Simulated_Interrupt
  (EIM_5, 250, 5, R_S'Access);

H_I : Interrupt_Handler
  (EIM_5, EIM_5_Priority, S_I'Access);

P_I : aliased Interrupt_Server_Parameters :=
  (Pri    => EIM_5_Priority,
   Period => Milliseconds (25),
   Budget => Milliseconds (2));

E_I : Deferrable_Interrupt_Server (P_I'Access);
```

The execution-time of all tasks and interrupt levels were polled every major period using a timing-event and sent on the USART by a background task. Utilization statistics based on 25000 samples are listed in Table 2.

Table 2. Utilization statistics

Task	Max	Min	Mean	Svar
I	0.0676	0.0372	0.0521	0.0041
S	0.3915	0.2268	0.3322	0.0240
A	0.0732	0.0608	0.0670	0.0022
B	0.0969	0.0746	0.0858	0.0056
C	0.1983	0.1508	0.1745	0.0168
D	0.0992	0.0754	0.0873	0.0118

As seen when comparing Table 2 to Table 1 no task exceeds it budget. Recall that if any task exceeded its budged or failed to meet its deadline an exception would occur. When the application was run using release mechanisms without overrun detection task D lost its deadline after a few major periods.

The test results are not exactly reproducible due to the external interrupt signal being generated asynchronously with the main system. However replication of the test show only minor variance in the result.

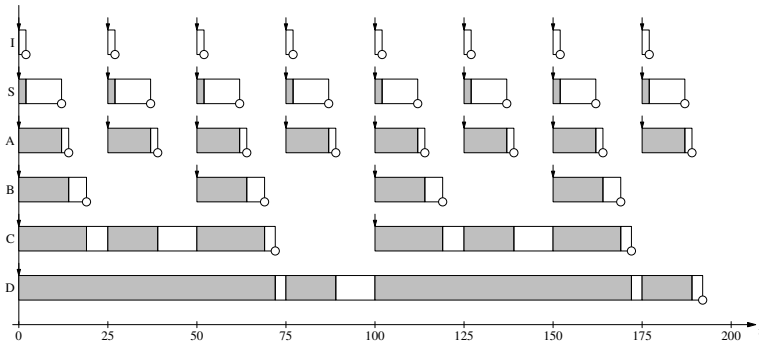

Figure 2. Gantt chart for application task set

6. Discussion

6.1. Framework

With the exception of the interrupt handling part the framework described in this paper has fewer features than the original framework due to the limitations of the Ravenscar profile. While the original framework takes advantage of the full Ada 2005 tasking model, the described framework cannot use mechanisms such as asynchronous control, requeue and select statements, and dynamic priorities. Instead it has to rely on tasks voluntarily giving up the processor after an overrun within the specified recovery time. It is not specified in which way overruns are to be handled.

Due to the limitations of the Ravenscar profile and the run-time environment execution-time servers could not be implemented. This limits the possibility of implementing aperiodic tasks, as their execution-time has to be controlled. However, it is possible to implement an aperiodic release mechanism with a built-in execution-time server. This scheme relies on the task blocking itself after a procedure `Hold` is called, and resuming execution when `Continue` is called. This may be done by the aperiodic task polling a flag and suspend on a suspension object. The `Overrun` procedure will be called if the aperiodic task fails to suspend within the recovery time.

Flexible interrupt handling is supported by the framework by specifying an interface for interrupt states and an underlying interrupt handling mechanism, similar to the task state and underlying real-time tasks. This allows for complex interrupt handlers. However, the sporadic interrupt release mechanism is also implemented as in the original framework as it is more efficient for interrupts simply releasing tasks.

The execution-time server for interrupt levels depends on the special interrupt level execution-time timers implemented on the AVR32 GNAT bare-board run-time environment [11]. This feature is therefore architecture specific. The interrupt server blocks all interrupts of an interrupt level if the execution-time budget is extinguished. This allows applications to limit the execution-time spent on interrupts and protect against burst of interrupts that would otherwise cause tasks to miss their deadlines.

The release mechanisms were implemented using a private protected object called through wrapper procedures implementing a limited interface instead of using synchronized interfaces due to the latter not being supported by the AVR32 run-time environment. This is not considered a problem as the wrapper procedures do not introduce a large overhead. Converting to using synchronized interfaces should be trivial as the private protected objects implement the operations of the interface. However, this would require the interface to include an initialization procedure as the requeue statement cannot be used to initialize the protected object on the first call.

6.2. Example application

The example application is not designed to be as realistic as possible but to demonstrate the usage of the real-time framework. Therefore the tasks execute a busy-wait for a given period instead of a real algorithm. This makes task execution-time easy to control. The tasks are programmed to overrun their budget 50% of the times executed, which is not very realistic. The high overrun frequency is chosen so that worst-case conditions occur more frequently.

The problem of execution-time overruns is solved by having the tasks poll a timeout flag set by the overrun procedure at regular intervals. While not an elegant solution, flag polling may be applicable to many systems as there will often be a loop or some recursive structure in the code executed by the real-time tasks. Alas, in order to use this solution the worst-case execution-time (WCET) between the pollings of the flags has to be known. The recovery time has to be greater than the sum of the WCET between the flag being polled and the WCET for exiting the task code and blocking on the release mechanism. Finding this WCET time may not be trivial. However it should be simpler than finding the WCET of the whole task, which again should lead to a less conservative budget.

There are also other possible ways of handling overruns than the method used by the example application. One possibility is to simply log the overrun and restart the system after a predefined number of overruns. Another possibility is to reconfigure the system after an overrun, choosing simpler algorithms for some or all of the tasks.

7. Conclusion

We have presented a new object-oriented real-time framework for Ada 2005 that is compliant with the Ravenscar profile. The framework allows periodic and sporadic tasks to be implemented as tagged types containing the task state and the code to be executed. The release mechanisms associated with the tasks support monitoring of execution-time overruns and deadline misses. By overriding procedures called in the case of execution-time overruns and deadline misses these events may be handled in an application-specific manner.

The framework also allows the execution-time spent on interrupt handlers of a given priority to be controlled by using the special AVR32 interrupt execution-time timers. When the execution-time budget for the interrupt priority is overrun all interrupts of that priority are disabled until the budget is replenished. The interrupt execution-time server is specific to the AVR32 architecture, while the rest of the framework is portable to other architectures and run-time environments.

The example application demonstrates how the framework may be combined with flag polling to handle execution-time overruns and avoid lower priority tasks missing their deadlines. Flag polling is used due to the restricted tasking model of the Ravenscar profile. While this scheme may not be as elegant as using asynchronous control it may easily be applied to algorithms with loops or recursion.

The described framework combined with the GNAT bare-board run-time environment provide an efficient object-oriented development platform which may reduce or remove an important class of software faults.

Acknowledgment

Many thanks to Andy Wellings and Alan Burns for allowing the use of their framework.

References

[1] ISO/IEC, *Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1*. [Online]. Available: http://www.adaic.com/standards/05rm/html/RM-TOC.html

[2] A. Burns, "The Ravenscar profile," *Ada Lett.*, vol. XIX, no. 4, pp. 49–52, 1999.

[3] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[4] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[5] J. A. de la Puente and J. Zamorano, "Execution-time clocks and Ravenscar kernels," *Ada Lett.*, vol. XXIII, no. 4, pp. 82–86, 2003.

[6] A. Wellings and A. Burns, *Ada-Europe 2007*. Springer Berlin / Heidelberg, 2007, ch. Real-Time Utilities for Ada 2005, pp. 1–14.

[7] Atmel Corporation, *AVR32 - Architecture Document*.

[8] ——, *AT32UC3A Series - Preliminary Datasheet*.

[9] J. F. Ruiz, "GNAT pro for on-board misson-critical space applications," *Ada-Europe*, 2005.

[10] K. N. Gregertsen and A. Skavhaug, "An efficient and deterministic multi-tasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller," in *Proc. DATE'09*, April 2009, IP5-7.

[11] ——, "Implementing the new Ada 2005 timing event and execution-time control features on the AVR32 architecture," February 2009, submitted to Journal of Systems Architecture.

[12] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar profile in high integrity systems," *Ada Lett.*, vol. XXIV, no. 2, pp. 1–74, 2004.

[13] Atmel Corporation, *ATmega169p Preliminary*.

[14] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recepies in C++*. Cambridge University Press, 2002.

A.3 Article No. 3

K. N. Gregertsen and A. Skavhaug:

“Execution time control for interrupt handling”

Published in proceedings of IRTAW-14 [29].

Execution-time control for interrupt handling

Kristoffer Nyborg Gregertsen, Amund Skavhaug
Department of Engineering Cybernetics, NTNU
N-7491 Trondheim, Norway
{gregerts,amund}@itk.ntnu.no

Abstract

This paper proposes that execution-time control features for interrupt handling should be added to the Ada standard library. By measuring the execution-time for interrupts separately the accuracy of task execution-time measurement will be also improved. It is described how the proposed features were implemented for the GNAT bare-board Ravenscar run-time environment on the Atmel AVR32 architecture. Test results for the implementation and an example of usage are presented.

1 Introduction

The execution-time clocks and timers standardized with Ada 2005 allow the execution-time of tasks to be monitored and overruns to be handled [7]. This feature is useful as the worst-case execution-time (WCET) of tasks may be hard to compute [12] and may also be significantly higher than the average execution-time. By utilizing the new execution-time control features it is possible to use less conservative budgets and instead handle execution-time overruns dynamically by methods such as asynchronous control [11] in order to avoid deadline misses. It is also possible to implement execution-time servers using these features [3].

The Ada 2005 standard does not specify which task, if any, that is to be charged the execution-time of interrupt handlers [7]. Most implementations charges the running task the execution-time of the interrupt handler [9]. This leads to inaccuracy in task execution-time measurement, and the worst-case overhead of interrupt handling has to be added to all task budgets as it is not possible to predict which task will be interrupted. This inaccuracy was raised as an issue when implementations of the new Ada 2005 features were evaluated [10].

By separately measuring the execution-time for interrupt priorities this inaccuracy in task execution-time measurement is addressed. This would also allow execution-time control for interrupt handling, thereby making it possible to protect the system from burst of interrupts that could otherwise result in tasks missing their deadlines. This additional safety may be very useful for high-integrity real-time systems. A demonstration of these feature is developed for the Atmel AVR32 architecture [2].

The AVR32 is a brand new architecture developed by Atmel Norway in cooperation with the Norwegian University of Science and Technology (NTNU). The GNU Ada Compiler (GNAT) and a bare-board Ravenscar run-time environment (GNATforLEON) were ported to AVR32 [1] at NTNU [4]. GNATforLEON is based on the Open Ravenscar Kernel developed for the ERC32 and LEON space application processors that was integrated into the GNU Ada Run-time Library (GNARL) by Josẽ F. Ruiz at AdaCore [8]. The version ported to the AVR32 did not include the Ada 2005 real-time features. These features were later implemented in GNATforLEON 1.3 [9]. This implementation charges the running task the execution-time of interrupt handlers.

The Ada 2005 real-time features were implemented from scratch for the AVR32 architecture and the UC3 core [5]. Improved accuracy for execution-time measurement compared to GNATforLEON 1.3 was achieved by using separate execution-time clocks for each interrupt level and accounting for

the effects of executing entries by proxy. It should be noted that the Ravenscar profile does not allow execution-time timers. Therefore strict compliance with the profile is broken by including this feature.

In the following there is a short introduction to the AVR32 architecture before the proposed additions to Ada standard library are presented. Next follows a description of how these features were implemented on the AVR32 UC3 core and test results for this implementation. An example of how the features may be used is given. Finally there is a discussion of the value added by the proposed features, the additions to the standard, the AVR32 implementation, and the portability of the features.

2 The AVR32 architecture

The Atmel AVR32 [2] is a 32-bit RISC architecture optimized for high code density and high computational throughput with low power consumption. The register file consists of 13 general purpose registers (R0 to R12), the link register (LR) used for storing the routine return address, the program counter (PC) and the system register (SR). The AVR32 has four interrupt levels, a Non-Maskable Interrupt (NMI) and exceptions.

The UC3 core [1] used in this paper is the second implementation of the AVR32 architecture. It is primarily intended for embedded control applications where deterministic execution-time is paramount. It has a three-stage pipeline integrated with an internal SRAM that bypasses the system bus, allowing deterministic, single-cycle read/write memory access. The core is shown in Figure 1.

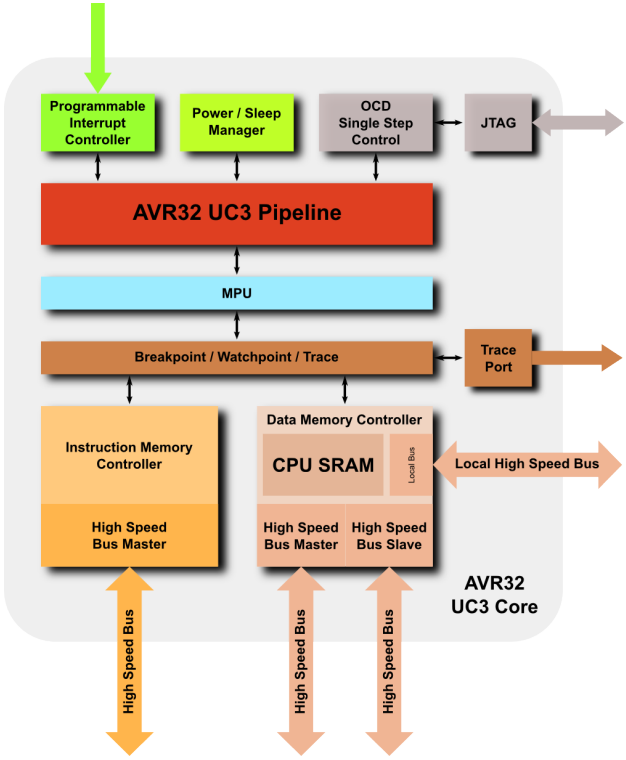

Figure 1: The AVR32 UC3 core

The UC3 has a programmable interrupt controller making it possible to configure the priority of interrupt groups. Prior to entering an interrupt level the UC3 automatically pushes registers R8 through R12, PC, LR and the system register on the stack.

The 32-bit COUNT/COMPARE system registers are used for execution-time measurement. The COUNT register has the value 0 at start-up. It is incremented on every clock cycle of the CPU and overflows silently. The COMPARE register is used for setting off the COMPARE interrupt when COUNT equals COMPARE. The interrupt is disabled when COMPARE has the value 0, which is the start-up value of the register. The interrupt is cleared by writing to the COMPARE register.

3 Standard library modifications

In order to support interrupt execution-time measurement and timer some additions had to be made to Annex D of the Ada 2005 standard [7]. It was chosen to make these changes in the existing execution-time packages specified in D.14 and D.14.1 instead of adding new packages to the standard library.

3.1 Execution-time measurement

In `Ada.Execution_Time` specified in section D.14 a function was added to support execution-time measurement for interrupt priorities:

```
function Interrupt_Clock
  (Priority : System.Interrupt_Priority)
  return CPU_Time;
```

This function returns the total execution-time spent by all interrupt handlers of the given interrupt priority since system start-up. It may be thought of as the execution-time clock of a pseudo task serving all interrupts of that interrupt priority.

3.2 Execution-time timers

In `Ada.Execution_Time.Timers` specified in section D.14.1 a tagged type `Interrupt_Timer` with the interrupt priority as discriminant that inherits `Timer` was defined for supporting execution-time timers for interrupt priorities:

```
Pseudo_Task_Id : aliased constant
  Ada.Task_Identification.Task_Id
  := Ada.Task_Identification.Null_Task_Id;

type Interrupt_Timer
  (I : System.Interrupt_Priority)
   is new Timer (Pseudo_Task_Id'Access) with private;
```

None of the operations of `Timer` are overridden as it is assumed that the same underlying mechanism will be used both for task and interrupt timers and that the only reason for having a separate type for interrupt timers is the difference in the discriminant.

An access to the constant `Pseudo_Task_Id` with the value `Null_Task_Id` is used as the discriminant `T` for all interrupt timers. This violates the Ada 2005 standard which states that a program error is to be raised for all operations on `Timer` if the value of `T.all` is `Null_Task_Id`. It was however needed to do this in order to define the `Interrupt_Timer` type.

4 AVR32 implementation

The execution-time control features were implemented in a new package `System.BB.TMU` (*Time Management Unit*) of the bare-board run-time environment.

The package defines the type `CPU_Time` representing execution-time as a 64-bit modular integer, a limited private type `Timer` used both for execution-time measurement and timers, an access type `Timer_Id` for the pointing to timers, and a procedure access type `Timer_Handler` taking an address as argument for low-level timer handlers. The type `Timer` is defined in the private part of the package as:

```
type Timer is
   record
      Active_TM : Timer_Id;
      Base_Time : CPU_Time;
      pragma Volatile (Base_Time);
      Timeout   : CPU_Time;
      Handler   : Timer_Handler;
      Data      : System.Address;
      Active    : Boolean;
      Acquired  : Boolean;
    end record;
```

Internally the TMU package may be though of as having two layers, a high-level layer managing timers, making sure that the correct timer is always *active*, and a low-level layer performing operations such as swapping timers and measuring execution-time.

4.1 Timer management

There are several possible active timers in the context of the running task since there are separate timers for the pseudo tasks such as the idle task and the interrupt server tasks, and the timer of another task is to be active when executing an entry by proxy. The possible timers in the context of a task and the relation among them are shown in Figure 2. The figure is simplified by only showing an arrow representing interruption of one level by the next one. When traversing the graph the history is kept so that the correct timers may be restored later.

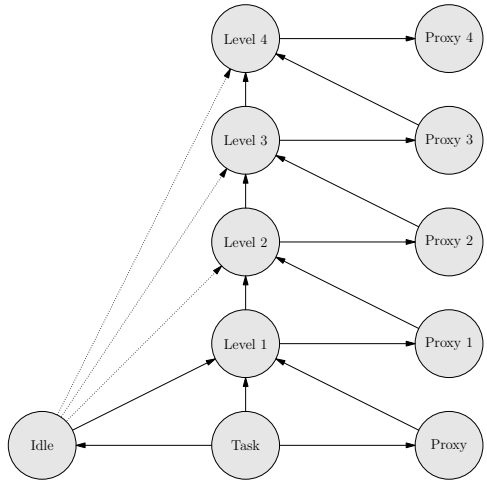

Figure 2: Possible timers of a task

After system initialization there is one and only one active timer at any time indicated by the field `Active` of the timer being set. The first active timer is that of the environment task. If the running task is not executing an interrupt, the idle loop, or an entry by proxy then the timer of the running task is active.

A stack of timer accesses is used for keeping track of the different interrupt level timers. The bottom element of the stack always points to the timer of the running task and is updated when a context switch is executed. The stack is pushed and popped when moving in the vertical direction of Figure 2, entering and leaving interrupt levels.

The field `Active_TM` is used for keeping track of horizontal movement in Figure 2 pointing to another timer when executing an entry by proxy or the idle loop.

4.2 Execution-time measurement

Execution-time is measured as the number of CPU clock cycles used since start-up. The `Base_Time` of a timer used for keeping track of absolute execution-time, also referred to as t_{base}, is initially zero. The execution-time for an active timer is the sum of the base time and the value of the COUNT system register, referred to as c:

$$t = t_{\text{base}} + c, \text{ where } c \in \{0, 1, \ldots, 2^{32} - 1\} \tag{1}$$

When a timer is deactivated its base time is updated:

$$t_{\text{base}} \leftarrow t_{\text{base}} + c \tag{2}$$

The execution-time measurement depends on the CPU counter never overflowing. To prevent this the value C written to COMPARE is never greater than a constant C_{max} chosen to be $2^{31} - 1$. When COUNT equals this value a COMPARE interrupt will be pending causing the timer to be inactivated and its base time updated when the interrupt is handled. The time-line for execution-time measurement is shown in Figure 3.

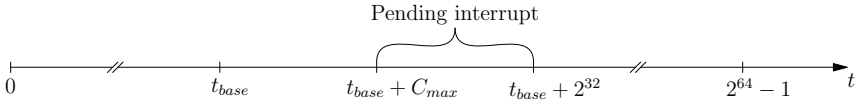

Figure 3: Execution-time for a timer from 0 to the final overflow.

4.3 Execution-time events

Execution-time events for timers are supported by having a `Timeout` field also referred to as t_{timeout} specifying when the timer is to expire, an access to a procedure `Handler` that is to be called when the timer expires, and an address `Data` to be given as an argument when calling the handler.

The COMPARE register is updated with the value C when timers are swapped, an active timer is set or an active and set timer is cancelled. If the timer is cleared C is set to C_{Max}, otherwise C is computed from the difference d between t_{timeout} and t_{base} by the following equation:

$$C = \begin{cases} 1 & \text{if } d < 1, \\ d & \text{if } 1 \leq d \text{ and } d \leq C_{max}, \\ C_{max} & \text{if } d > C_{max} \end{cases} \tag{3}$$

The handler for the COMPARE interrupt has the highest interrupt priority. When executed the timer of the highest interrupt priority is active and on the top of the stack, the timer causing the interrupt is pointed to by `Active_TM` of the timer below on the stack. The handler gets this timer and checks if it has expired as false interrupts may occur due. If expired the timer is cleared and the handler called with `Data` as argument. No further action is taken in the case of a false interrupt.

4.4 CPU counter primitives

Four primitive operations to manipulate the COUNT and COMPARE system registers were added to the architecture specific `CPU_Primitives` package:

- The function `Get_Count` that returns the value of COUNT system register.
- The procedure `Reset_Count` that resets COUNT and sets the COMPARE register to the value C given as argument without causing any false interrupts.
- The procedure `Swap_Count` that performs the same operation as `Reset_Count` but also returns the previous value of COUNT incremented in order to avoid execution-time leakage.
- The procedure `Adjust_Compare` that changes the value of COMPARE to C without losing an interrupt. If the procedure is called with a C that is less than COUNT, the COMPARE interrupt will be pending when leaving the procedure.

All these routines were written as inline assembler code for maximal control and efficiency.

4.5 Application Programming Interface

The tagged type `Timer` visible through the package `Ada.Execution_Time.Timers` includes a `Timer_Id` pointing to the associated kernel timer and a field holding the current handler.

The `Timer_Id` is initialized the first time `Set_Handler` is called. The initialization procedure acquires the kernel timer for an interrupt priority if the timer is in `Interrupt_Timer'Class` or else the timer for the task ID. If the timer is already acquired or not available a `Timer_Resource_Error` is raised.

When the kernel timer is set, the `Data` address argument is the address of the timer object and the `Handler` an access to an internal procedure. When called by the COMPARE handler this procedure type-casts the argument `Data` to an timer access before calling the handler provided by the programmer. This resembles the use of void pointers in C, and may be seen as a breach of Ada programming practice, but was chosen for reasons of simplicity and efficiency.

All operations performed on timers objects are done atomically by using the kernel protection package.

5 Tests

There is a non-zero overhead to the execution-time measurement of a task being interrupted or a task executing an entry by proxy. Test programs were made to measure this overhead.

The Atmel EVK1100 evaluation board with the UC3A0512 microcontroller [1] clocked at 60 MHz was used for the tests. Results were sent on the USART channel from the EVK1100 to a PC where it was retrieved and analyzed. The Atmel JTAG ICE Mk II was used for programming and debugging the UC3.

5.1 Cost of interruption

This program has a single loop executed by the environment task where the execution-time is first measured using `Clock`, then a busy wait for 50 ms is executed before the execution-time is measured again. The two measured execution-times are sent to the PC and the loop is repeated.

The only active interrupt in this program is the real-time clock overflow that will occur at most once between the two measurements of execution-time. The difference between the shortest and longest measured execution-time between the clock calls should give an indication on the overhead of interruption.

When the test was run and 5000 samples were gathered, there were only three unique values of the difference between the clock before and after executing the busy wait: 600050, 600182 and 600183.

This is a testament of the deterministic nature of the AVR32 UC3 microcontroller. It is inferred that the interruption overhead is 133 CPU clock cycles.

5.2 Cost of executing an entry by proxy

This test program has two tasks named A and B and a protected object. The protected object has an entry with a guard and a procedure that opens that guard. The protected object has the highest priority in order to avoid interruption while executing its operations.

Task A has higher priority than B and measures its execution-time before and after executing the protected procedure. After executing the procedure, A sends its execution-time measurements to the PC before it is delayed until next release. Task B will simply call the entry and then be delayed until next release. By releasing A after B half of the times, A will call the procedure both when B is blocked on the entry, in which case the entry will be executed by proxy, and when B is not blocked.

When running the test gathering 5000 samples, the execution-time between reading the clock was measured to be 387 CPU cycles when no entry by proxy was executed and 610 when the entry was executed, giving a cost of 223 CPU cycles. As with the previous test the unique values are due to the determinism of the AVR32 UC3 microcontroller. However 223 CPU cycles is the best-case cost, the worst-case situation is when task B has a handler set long into the future due to the calculation of the COMPARE value C for task B when swapping timers. When task B has a timer set to expire at `CPU_Time'Last`, the cost was measured to be 237 CPU cycles.

6 Application

Extensions were be made to the real-time framework by Andy Wellings and Alan Burns [11] to utilize the interrupt execution-time control features [6].

6.1 Interrupt handling

In order to support more flexible interrupt handling an interface for interrupt states was defined:

```
package Interrupt_States is
   type Interrupt_State is limited interface;
   procedure Handler
     (S : in out Interrupt_State) is abstract;
   procedure Enable
     (S : in out Interrupt_State) is abstract;
   procedure Disable
     (S : in out Interrupt_State) is abstract;
   type Any_Interrupt_State is
     access all Interrupt_State'Class;
end Interrupt_States;
```

Device drivers for peripherals associated with an interrupt may for instance implement procedures for enabling and disabling the interrupt while the application has a type that inherits the device driver and implements the interrupt handler itself.

A protected object for handling interrupts was defined as:

```
protected type Interrupt_Handler
  (Id  : Interrupt_Id;
   Pri : Interrupt_Priority;
   S   : Any_Interrupt_State) is
   pragma Interrupt_Priority (Pri);
private
   procedure Handler;
   pragma Attach_Handler (Handler, Id);
end Interrupt_Handler;
```

The private handler of the protected objects simply executes a dispatching call to the handler of the interrupt state.

6.2 Interrupt servers

Interrupt servers take advantage of the interrupt timers to control the execution-time spent on handling interrupts of a given interrupt priority. An interface for these servers was defined as:

```
package Interrupt_Servers is
   type Interrupt_Server_Parameters is
      record
         Pri : Interrupt_Priority;
         Budget : Time_Span;
         Period : Time_Span;
      end record;
   type Interrupt_Server is synchronized interface;
   procedure Register
     (S : in out Interrupt_Server;
      I : Any_Interrupt_State) is abstract;
   type Any_Interrupt_Server is
     access all Interrupt_Server;
end Interrupt_Servers;
```

The interrupt server parameters give the interrupt priority, budget and replenishing period of the interrupt server. The synchronized interface `Interrupt_Server` has a single operation `Register` for registering interrupts to be controlled.

The type `Deferrable_Interrupt_Server` implements the interface `Interrupt_Server` and was defined as:

```
protected type Deferrable_Interrupt_Server
  (Param : access Interrupt_Server_Parameters)
is new Interrupt_Server with
   procedure Register (I : Any_Interrupt_State);
   pragma Interrupt_Priority (Any_Priority'Last);
private
   procedure Replenish (TE : in out Timing_Event);
   procedure Overran (TM : in out Timer);
   Replenish_Event : Timing_Event;
   Execution_Timer : access Interrupt_Timer;
   Next : Time;
   Disabled : Boolean := True;
   Registered : Natural := 0;
   States : State_Array;
end Deferrable_Interrupt_Server;
```

The interrupt execution-time is controlled by using the interrupt timer of the given level and a timing event replenishing the budget. The first call to `Register` initializes the object and sets the first replenish event. All registered interrupts are disabled on overrun and enabled every time the budget is replenished:

```
procedure Replenish (TE : in out Timing_Event) is
begin
   Execution_Timer.Set_Handler
     (Param.Budget, Overran'Access);
   if Disabled then
      Disabled := False;
      for I in 1 .. Registered loop
         States (I).Enable;
      end loop;
   end if;
   Next := Next + Param.Period;
   TE.Set_Handler (Next, Replenish'Access);
end Replenish;
```

```
procedure Overran (TM : in out Timer) is
begin
   if not Disabled then
      Disabled := True;
      for I in 1 .. Registered loop
         States (I).Disable;
      end loop;
   end if;
end Overran;
```

Interrupts are assumed to be initially disabled and are enabled on the first replenish event.

6.3 Usage

In order to use the interrupt system the programmer could define a tagged type `My_Interrupt` extending the AVR32 device driver `External_Interrupt` which implements the interface `Interrupt_State`. The device driver implements the procedures to enable, disable and clear the external interrupt identified by its discriminant, so `My_Interrupt` only needs to implement the interrupt handler:

```
procedure Handler (S : in out My_Interrupt) is
begin
   S.Clear;
   --  Do work of handler...
end Handler;
```

The interrupt state, an interrupt handler and an interrupt server may be declared as:

```
S_I : aliased My_Interrupt (EIM_5);

H_I : Interrupt_Handler
  (EIM_5, EIM_5_Priority, S_I'Access);

P_I : aliased Interrupt_Server_Parameters :=
  (Pri    => EIM_5_Priority,
   Period => Milliseconds (25),
   Budget => Milliseconds (2));

E_I : Deferrable_Interrupt_Server (P_I'Access);
```

The interrupt state `S_I` needs to be registered to the interrupt server `E_I` when the application is initialized.

7 Discussion

7.1 Interrupt execution-time control

When charging the interrupted task the execution-time of interrupt handlers the WCET of all possible interrupt handler invocations have to be added to the execution-time budget of all tasks as there is no way of predicting how many times each task will be interrupted. This leads to reduced CPU utilization. By not charging the interrupted tasks the execution-time of interrupt handlers the accuracy of execution-time measurement is improved and the task budgets can be tighter. Ideally there should be zero execution-time cost to the interrupted task.

The approach taken by this paper is to charge the execution-time of interrupt handlers to a pseudo task though to execute all interrupt handlers of a given interrupt priority. This allows the execution-time spent on handling interrupts of a given task to be retrieved, which may be useful for debugging purposes, but more importantly it allows execution-time timers for interrupt handling.

By using interrupt timers it is possible to implement execution-time servers for interrupts priorities making it possible to protect the system against bursts of interrupts that would otherwise result in system failure. Such an execution-time server has been implemented in a real-time framework based

on work by Andy Wellings and Alan Burns [6, 11]. It should be noted that system interrupts such as the clock interrupt, timing event interrupt and the interrupt for execution-time should never be disabled as this could cause run-time environments to malfunction.

An alternative approach would be to not charge any task the execution-time of interrupt handlers. This would give the same improvement in execution-time measurement for tasks without any additions to the Ada standard library, but would obviously not allow execution-time control for interrupt handling. The additional safety provided by the having execution-time timers for interrupt priorities seems to justify the modifications to the Ada standard library.

7.2 Standard library modifications

The proposed changes to the Ada standard library are in the existing packages `Ada.Execution_Time` and `Ada.Execution_Time.Timers`. Changes were made to existing packages instead of adding new ones since the additions are few and fairly simple. However it might be argued that the interrupt execution-time control features should be isolated in child packages so that architectures were these features are not implementable may simply leave these packages out.

The package `Ada.Execution_Time` is modified by adding a function `Interrupt_Clock` for retrieving the execution-time spent on handling interrupts of the given interrupt priority. The function works in the same way as `Clock` would for an interrupt pseudo task serving all interrupts of priority. An alternative would be to measure the execution-time spent handling each type of interrupt. This approach would require more run-time support, but would be more suitable for multiprocessors where several interrupts of the same priority may be handled at the same time in which case the pseudo task approach taken is this paper would not be possible.

The package `Ada.Execution_Time.Timers` is modified by adding the aliased constant `Pseudo_Task_Id`, and the tagged type `Interrupt_Timer` inheriting `Timer` taking the interrupt priority as discriminant, and using `Pseudo_Task_Id` as the discriminant `T` for `Timer` indicating that it is not bound to an ordinary task. This conflicts with the Ada 2005 standard as the `Pseudo_Task_Id` has the value `Null_Task_Id`. An alternative would be to have a special `Task_Id` defined for each pseudo interrupt server task, and use the type `Timer` for both ordinary task timers and interrupt timers. However this approach was not used as other parts of the run-time environment then would have to include checks for the special `Task_Id`.

7.3 AVR32 implementation

The Ada 2005 real-time features were implemented from scratch on the Atmel AVR32 architecture [5]. The addition of interrupt execution-time control and taking into account the effects of entry-by-proxy execution is the main difference between this implementation and GNATforLEON 1.3 [9].

The AVR32 implementation is not ideal with regards to interrupt execution-time measurement as there is an overhead to the interrupted task due to the timers being switched by software. The cost of interruption by the clock interrupt was found by testing to be 133 CPU cycles. This the best-case cost for general interrupts as the highest level interrupt timer is never set with a handler. The additional cost of being interrupted by an interrupt timer with a handler set long into the future is 14 CPU cycles as observed with execution of entries by proxy, giving a worst-case cost of 147 CPU cycles.

Execution by proxy improves system performance by reducing the number of context switches needed. This implementation always charges the blocked task the execution-time spent on the entry thereby improving the accuracy of the execution-time measurement as this execution-time does not have to be added to the budget of the task executing the entry by proxy. The usefulness of this feature is however highly dependent of the implementation overhead of changing timers compared to the execution-time of the entry. As seen from the test results the worst-case cost of executing an entry by proxy is 237 CPU cycles, which definitely is more than the cost of executing an entry with a null body. Still the benefit of having a known constant worst-case cost instead of an unknown cost when executing entries by proxy seems to justify the overhead.

7.4 Portability and hardware support

The implementation of the proposed interrupt execution-time control features were done on the AVR32 UC3 microcontroller. Unfortunately recent changes to the COUNT / COMPARE semantics [1] for the UC3 core has made the registers less useful for execution timing. The new semantics state that when COUNT equals COMPARE the interrupt line should be asserted and COUNT set to 0. The original semantics were used in this paper since only the engineering presample version was available with the EVK1100 at the time of development. The implementation could be adapted to the new semantics by adding more logic to the CPU primitives.

The bare-board GNAT implementation of the Ada 2005 execution-time features is portable to other architectures as long as it is possible to implement the CPU primitives with fairly low overhead. The features should also be portable to other embedded run-time systems that allow direct control of the timer hardware. In order to port the features to operating systems such as Linux, the kernel would have to be modified to support interrupt execution-time measurement. This would be harder, but fully possible.

8 Conclusion

The main contributions of this work is an extension to the Ada 2005 standard library allowing execution-time control for interrupt handling, and the implementations of these features on the Atmel AVR32 architecture. The benefits of this are twofold:

1. The accuracy for task execution-time measurement is improved as the execution-time of interrupt handlers is not charged the interrupted task. This allows task budgets to be tighter, and thereby higher CPU utilization.

2. It is possible to monitor and control the execution-time spent on handling interrupts of a given priority using execution-time servers. This makes it possible to protect the systems from bursts of interrupts that could otherwise result in tasks missing their deadline.

In the authors opinion the proposed features would be a valuable addition to the Ada programming language and should be particularly useful for high-integrity real-time systems. Therefore the authors recommend that the features are added to the next revision of the Ada standard.

Acknowledgment

Thanks to Alan Burns for giving valuable advice on the Ada 2005 execution-time control features, and to Atmel Norway for providing the needed development hardware.

References

[1] Atmel Corporation. *AT32UC3A Series - Preliminary Datasheet.*

[2] Atmel Corporation. *AVR32 - Architecture Document.*

[3] A. Burns and A.J. Wellings. Programming execution-time servers in Ada 2005. In *Proc. 27th IEEE International Real-Time Systems Symposium RTSS '06*, pages 47–56, Dec. 2006.

[4] K.N. Gregertsen and A. Skavhaug. An efficient and deterministic multi-tasking run-time environment for ada and the ravenscar profile on the atmel avr ®32 uc3 microcontroller. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, pages 1572–1575, April 2009.

[5] Kristoffer Nyborg Gregertsen and Amund Skavhaug. Implementing the new Ada 2005 timing event and execution-time control features on the AVR32 architecture. Submitted to Journal of Systems Architecture, February 2009.

[6] Kristoffer Nyborg Gregertsen and Amund Skavhaug. A real-time framework for ada 2005 and the ravenscar profile. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 515–522, Aug. 2009.

[7] ISO/IEC. *Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1.*

[8] José F. Ruiz. GNAT pro for on-board misson-critical space applications. *Ada-Europe*, 2005.

[9] Santiago Urue na, José Pulido, José Redondo, and Juan Zamorano. Implementing the new Ada 2005 real-time features on a bare board kernel. *Ada Lett.*, XXVII(2):61–66, 2007.

[10] Andy Wellings. Implementation experience with Ada 2005. *Ada Lett.*, XXVII, no 2:59–60, 2007. session report.

[11] Andy Wellings and Alan Burns. *Ada-Europe 2007*, chapter Real-Time Utilities for Ada 2005, pages 1–14. Springer Berlin / Heidelberg, 2007.

[12] Reinhard Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

A.4 Article No. 4

K. N. Gregertsen and A. Skavhaug:

“Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture”

Published in Journal of Systems Architecture (JSA) [30].

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture

Kristoffer Nyborg Gregertsen *, Amund Skavhaug

Department of Engineering Cybernetics, Norwegian University of Science and Technology, N-7491 Trondheim, Norway

ARTICLE INFO

Article history:
Received 3 February 2009
Received in revised form 17 June 2010
Accepted 1 August 2010
Available online 19 August 2010

Keywords:
Ada 2005
Real-time
Execution time control
GNAT
Atmel AVR32

ABSTRACT

This paper describes how the new Ada 2005 timing event and execution time control features were implemented for the GNAT bare-board Ravenscar run-time environment on the Atmel AVR32 architecture. High accuracy for execution time measurement was achieved by accounting for the effects of interrupts and executing entries by proxy. The implementation of timing events was streamlined by using a single alarm mechanism both for timing events and waking up tasks. Test results on the overhead and accuracy of the implemented features are presented. While the implementation is for the AVR32, it may serve as a blueprint for implementations on other architectures. It is also discussed how the presented design could be transferred to other systems such as C/POSIX and RTSJ.

1. Introduction

The size and complexity of embedded software increases, motivating the use of high-level programming languages to master the complexity and reduce the number of programming faults. Ada is an ISO standard programming language [1] that was designed for use in high-integrity systems and has many safeguards against common programming faults. The language also has excellent support for development and maintenance of large applications by its notation of packages. Furthermore Ada has language support for tasking and a rich set of synchronization primitives, making even multi-tasking applications portable.

The latest version of the language standard is Ada 2005 [1] which is an amendment to the Ada 95 standard and not a full language revision. Ada 2005 makes significant improvements to Ada 95, especially within high-integrity and real-time systems [2]. Among the changes are improvements to the OO-model such as supporting prefix method call notation and Java-like interfaces, more flexible access types, enhanced structure and visibility control for packages, extensions to the standard library, and new tasking and real-time features [2].

This paper focuses on two of the new real-time features in Ada 2005: timing events and execution time control. Timing events allow protected procedures to be executed at a given time without the use of tasks and delay statements. This feature is useful for a wide range of purposes, for instance to replenish execution time budgets and handling deadline misses in real-time systems [3]. The execution time control feature allows the execution time of tasks to be monitored and overruns to be handled. This may simplify software development as the worst-case execution time (WCET) of tasks may be difficult to compute [4] and may also be significantly higher than the average execution time, especially for architectures using performance-enhancing techniques such as deep pipelines, cache, branch prediction and similar. Execution time control allows the programmer to use less conservative budgets and instead handle execution time overruns dynamically. A task that has exceeded its budget may be stopped by means of such as asynchronous control to avoid other tasks missing their deadlines [3]. It is also possible to implement execution time servers using group execution time budgets [5].

The Ada 2005 standard does not specify which task, if any, is to be charged the execution time of interrupt handlers [1]. Earlier implementations known to the authors charge the running task this execution time [6,7]. This leads to inaccuracy in execution time measurement, and the time budget of all tasks has to be augmented with the worst-case overhead of interrupt handling as it is not possible to predict which task will be interrupted. This inaccuracy was raised as an issue when implementations of the new Ada 2005 features were evaluated [8]. The implementation described in this paper addresses this issue by measuring the execution time of interrupt handlers separately for each interrupt priority instead of charging the interrupted task. This also allows execution time

* Corresponding author. Tel.: +47 48153380.
E-mail addresses: gregerts@itk.ntnu.no (K.N. Gregertsen), amund@itk.ntnu.no (A. Skavhaug).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.08.001

control for interrupt handling, making it possible to protect the system from bursts of interrupts, an error that cannot be prevented by design and that could result in deadlines being missed.

The timing event and execution time control features were implemented from scratch on the AVR32 UC3 microcontroller series [9]. The new AVR32 architecture [10] is developed by Atmel Norway in cooperation with the Norwegian University of Science and Technology (NTNU). The AVR32 was chosen as the close ties between NTNU and Atmel Norway make it possible to add and test new hardware features. The GNU Ada Compiler (GNAT) and the run-time environment were ported to AVR32 and the UC3 microcontroller series by the authors at NTNU making Ada available for this architecture [11]. The ported run-time environment is based on the Open Ravenscar Kernel (ORK) developed at the Technical University of Madrid (UPM) for the ERC32 and LEON processors used by the European Space Agency. This kernel was later integrated into the run-time library as GNATforLEON [12].

The Ravenscar profile defines a sub-set of the Ada tasking model designed to provide the static and deterministic environment needed in some high-integrity systems [13,14]. By removing complex tasking features the profile also allows more compact and efficient run-time environments to be developed. Execution time timers are not allowed by Ravenscar [1], so strict compliance with the profile is lost by implementing this feature. However this restriction is debated in the community. Research on execution time timers for Ravenscar kernels done at UPM [15] prior to the writing of the Ada 2005 standard concluded that these timers were useful and indeed compatible with the Ravenscar profile. The features were also implemented at UPM for the Open Ravenscar Kernel [16] and later ported to version 1.2 of GNATforLEON [7]. We consider the value added to the system by implementing execution time control greater than the value of retaining strict compliance with the Ravenscar profile.

In the following there is an introduction to the AVR32 architecture, the model of clocks and timers used in this paper, and the new Ada 2005 real-time features. Next follows a description of how timing events and execution time control were implemented on the AVR32 UC3 microcontroller, and test results for this implementation are presented. Finally there is a discussion of the implementation, possible applications of the implemented features, the portability of the implementation to other architectures, and how the design could be transferred to other languages.

2. The AVR32 architecture

The Atmel AVR32 [10] is a 32-bit RISC architecture optimized for high code density and high computational throughput with low power consumption. The architecture has a fairly small register file consisting of 13 general purpose registers (R0–R12), the link register (LR) used for storing the routine return address, the program counter (PC) and the system register (SR). The AVR32 has four interrupt levels, and exceptions such as the Non-Maskable Interrupt (NMI) and illegal address exception.

The UC3 core [9] used in this paper is the second implementation of the AVR32 architecture. It is primarily intended for embedded control applications where deterministic execution time is paramount. It has a three-stage pipeline integrated with an internal SRAM that bypasses the system bus, allowing deterministic, single-cycle read/write memory access.

The 32-bit COUNT/COMPARE system registers are used for execution time measurement. The COUNT register has the value 0 at start-up and is incremented on every clock cycle of the CPU. The COMPARE register is used for triggering the COMPARE interrupt when COUNT equals COMPARE. The interrupt is disabled when COMPARE has the value 0, which also is the start-up value. The interrupt is cleared by writing the COMPARE register.

3. Clocks and timers

A *clock* measures the passage of time using a physical process as reference, typically a crystal oscillator. Clocked time is discrete and represented by a count of ticks $c \in \mathbb{N}_0$. Each tick corresponds to a duration T, so the measured time is $t = T \cdot c$. Most clocks have inaccuracies caused by jitter and drift when compared to a reference clock. Jitter is the phenomenon such that the duration between the ticks is not a constant T but a stochastic function $\widehat{T}$. If the expected duration $E(\widehat{T})$ between the ticks is not equal to T the clock will also drift compare to a reference clock. This drift will accumulate over time. Such inaccuracies are not considered in this paper. Clocks are allowed to be stopped and resumed, but are required to be monotonic. Therefore the following relation holds between samples where t_i denotes the i'th sample of a clock:

$$t_1 \leqslant t_2 \leqslant \ldots \leqslant t_{i-1} \leqslant t_i \quad \forall i \in \mathbb{N}. \tag{1}$$

The real-time clock is used for system operations such as task release and setting task deadlines. There is only one instance of this clock. The clock is activated at system start-up, also called the epoch, and is never stopped. Execution time clocks are used to measure the total time an executable entity has been running on the system. There is one clock for each entity. The clock is started when the entity is scheduled for execution and stopped when it is suspended, blocked or terminated, or preempted by another entity. In this paper the executable entities considered are tasks and interrupt handlers.

A *timer* is associated with a clock and is used to generate an *event* that occurs when the clock reaches a specified time t_e. A timer is said to be *set* with an event. An event occurrence is handled at a time $t \geqslant t_e$, usually by a user-specified procedure to be called by the system kernel. Several timers may be associated with a single clock as seen in Fig. 1. Event occurrences for one clock are required to be handled in time order with earliest event first. The order for events occurring at the same time is not specified, but FIFO order will be used in this paper.

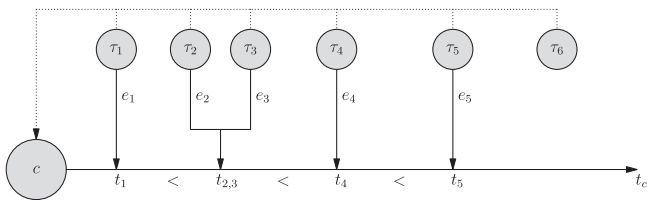

Fig. 1. Six timers associated with one clock. All timers but τ_6 are set with events. Timers τ_1 and τ_3 are set with events e_2 and e_3 occurring at the same time $t_{2,3}$. Assuming that FIFO order is used and τ_2 was set before τ_3 the events will be handled in the order e_1, e_2, e_3, e_4 and e_5.

```
package Ada.Real_Time.Timing_Events is

  type Timing_Event is tagged limited private;
  type Timing_Event_Handler
       is access protected procedure (Event : in out Timing_Event);

  procedure Set_Handler (Event   : in out Timing_Event;
                         At_Time : in Time;
                         Handler : in Timing_Event_Handler);
  procedure Set_Handler (Event   : in out Timing_Event;
                         In_Time : in Time_Span;
                         Handler : in Timing_Event_Handler);
  function Current_Handler (Event : Timing_Event)
       return Timing_Event_Handler;
  procedure Cancel_Handler (Event     : in out Timing_Event;
                            Cancelled : out Boolean);

  function Time_Of_Event (Event : Timing_Event) return Time;

private
  ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

Listing 1. Timing events definition.

4. The new Ada 2005 real-time features

4.1. Timing events

Timing events allow protected procedures to be called at a specified time without the need for a task or delay statement to control their activation. The package used for timing events is defined in section D.15 of the standard as shown in Listing 1.

A Timing_Event object is said to be set if associated with a non-null handler and cleared otherwise. All timing event objects are initially cleared. The type Timing_Event_Handler identifies a protected procedure that is to be executed when the timing event occurs. The timing event is cleared before the handler is called. There are two procedures for setting a timing event with a handler, both named Set_Handler: one taking the absolute time of the event and the other a relative time. If Set_Handler is called for an already set event, the handler is replaced. If called with a null handler the event is cleared. Handlers may be cancelled using Cancel_Handler which returns whether the handler was cancelled or not. The function Current_Handler returns the current handler of the event, while the function Time_Of_Event returns the time of the event.

Implementations are required to perform operations on a timing event object atomically. Implementations are also required to document the upper bound on the overhead of the handler being called. The Ravenscar profile only allows timing events declared at library level.

4.2. Execution time control

There are three packages associated with execution time control, Ada.Execution_Time that defines the types used for execution time measurement and execution time clocks, and two child

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is

   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant := implementation-defined-real-number;
   CPU_Tick : constant Time_Span;

   function Clock
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;

   function "+"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "+"  (Left : Time_Span; Right : CPU_Time) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : CPU_Time)  return Time_Span;

   function "<"  (Left, Right : CPU_Time) return Boolean;
   function "<=" (Left, Right : CPU_Time) return Boolean;
   function ">"  (Left, Right : CPU_Time) return Boolean;
   function ">=" (Left, Right : CPU_Time) return Boolean;

   procedure Split
     (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

   function Time_Of (SC : Seconds_Count;
                     TS : Time_Span := Time_Span_Zero) return CPU_Time;

private
   ... -- not specified by the language
end Ada.Execution_Time;
```

Listing 2. Execution time definition.

packages Timers and Group_Budgets used for handling execution time overruns of single tasks and groups of tasks respectively. Group budgets are not implemented in this work for reasons of efficiency, and will therefore not be described any further here.

The execution time package is defined in section D.14 of the standard as shown in Listing 2. The type CPU_Time represents the execution time of a task, which is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf [1, D.14]. It is implementation defined which task, if any, is charged the execution time used by interrupt handlers and run-time services on behalf of the system. The package defines the function clock for getting the execution time of a task.

The child package for execution time timer is defined in section D.14.1 of the standard as shown in Listing 3. An object of tagged type timer represents the source of an execution time event for a single task and is capable of detecting execution time overruns. As for timing events a timer is said to be set if associated with a non-null handler and cleared otherwise, and all timers are initially cleared. The type Timer_Handler identifies a protected procedure to be executed when the timer *expires*. Timers are set and cancelled as with timing events with the exception of the absolute time for Set_Handler being given as CPU_Time. The function Time_Remaining replaces Time_Of_Event and returns the time remaining until the timer expires.

As with timing events the implementation is required to perform the operations on a timer object atomically. Implementations are allowed to limit the number of timers possible for a single task. If this number of timers is exceeded then Timer_Resource_Error is raised. In this work there is a limit of one timer for each task for reasons of efficiency. This limitation was also recommended for use with the Ravenscar profile [15]. As stated earlier the Ravenscar profile does not allow this feature.

4.3. Standard library modifications

In order to support execution time control for interrupts some additions had to be made to Annex D of the Ada 2005 standard [1]. It was decided to make these changes in the existing execution time packages specified in D.14 and D.14.1 instead of adding new packages to the standard library.

The two functions shown in Listing 4 were added to Ada.Execution_Time in order to support execution time measurement for the system idle task and interrupt priorities. The function Idle_Clock returns the execution time the system has been idling since start-up, while Interrupt_Clock returns the total time spent by all interrupt handlers of the given interrupt priority since start-up.

The tagged type Interrupt_Timer shown in Listing 5 was added to Ada.Execution_Time.Timers shown in Listing 3 to support execution time timers for interrupt priorities. The type takes the interrupt priority as discriminant and inherits Timer. None of the operations of Timer are overridden as it is assumed that the same underlying mechanism will be used both for task and interrupt timers and that the only reason for having a separate type for interrupt timers is the difference in the discriminant.

An access to the constant Pseudo_Task_Id with the value Null_Task_Id is used as the discriminant T for all interrupt timers. This violates the Ada 2005 standard which states that a program error is to be raised for all operations on Timer if the value of T.all is Null_Task_Id. It was however needed to do this in order to define the Interrupt_Timer type.

```
with System;
package Ada.Execution_Time.Timers is

   type Timer (T : not null access constant
                   Ada.Task_Identification.Task_Id) is
      tagged limited private;

   type Timer_Handler is
      access protected procedure (TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority :=
   implementation-defined;

   procedure Set_Handler (TM      : in out Timer;
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   function Current_Handler (TM : Timer) return Timer_Handler;
   procedure Cancel_Handler (TM        : in out Timer;
                             Cancelled :    out Boolean);

   function Time_Remaining (TM : Timer) return Time_Span;

   Timer_Resource_Error : exception;

private
   ... -- not specified by the language
end Ada.Execution_Time.Timers;
```

Listing 3. Execution time timers definition.

```
function Idle_Clock return CPU_Time;

function Interrupt_Clock
  (Priority : System.Interrupt_Priority)
  return CPU_Time;
```

Listing 4. Additions to execution time package.

```
Pseudo_Task_Id : aliased constant Ada.Task_Identification.Task_Id
  := Ada.Task_Identification.Null_Task_Id;

type Interrupt_Timer (I : System.Interrupt_Priority)
   is new Timer (Pseudo_Task_Id'Access) with private;
```

Listing 5. Additions to execution time timers package.

5. Implementation of timing events

5.1. Design

The timing services of the bare-board run-time environment are based on the use of two hardware timers named *clock* and *alarm*. On the AVR32 two channels of the Timer/Counter module are used for these timers. The clock timer keeps the least significant part (LSP) of the system clock. On overflow the most significant part (MSP) of the clock resident in system memory is incremented by the clock interrupt handler. The alarm timer is used for setting off alarm interrupts within clock periods. Before implementing timing events the clock interrupt was handled before the alarm interrupt if both interrupt lines were asserted.

Prior to implementing Ada 2005 timing events the alarm timer was used exclusively for waking up delayed tasks. The functionality needed for this was spread over three kernel packages. The queue of future alarms for waking up tasks was managed as a linked list in the package System.BB.Threads.Queues. The type Thread_Descriptor of System.BB.Threads included fields for the alarm timeout and linking to the next alarm in the alarm queue. Setting the hardware alarm timer, handling the clock interrupt, and waking up delayed tasks were all done in the package System.BB.Time. When implementing timing events it was decided to use one alarm mechanism for both waking up tasks and handling timing events, and to gather all the functionality needed in System.BB.Time. The new relationship between these packages are shown in Fig. 2.

The alarm mechanism is represented by the type Alarm_Descriptor shown in Listing 6 that was added to the package Time. When calling the routines of the package the access type Alarm_Id is used. This follows the naming pattern of the package Threads where the types Thread_Descriptor and Thread_Id are used for kernel threads.

The type Alarm_Handler is an access to procedure taking System.Address as argument. The procedure is called when the alarm expires. This way of passing data resembles the use of void pointers in C and might be seen as a breach of good Ada programming practice, but was chosen for reasons of performance and simplicity. The procedure Set_Handler sets an alarm with a handler to be called at a given time with the given address passed as argument. The procedure Cancel_Handler for cancelling alarms and the function Time_Of_Alarm to get the time when an alarm expires are also defined.

5.2. Alarm queue

Internally there is a queue of pending alarms organized as a doubly linked list with a sentinel alarm at the end as shown in Fig. 3. The alarms in the queue are sorted in ascending order according to the value of the Timeout field. The first element of the queue is pointed to by the access First_Alarm while the sentinel at the end of the queue is named Last_Alarm. The sentinel is set to expire at Time'Last and is not associated with a handler since it is assumed that this time will never be reached.

When an alarm is set using the procedure Set_Handler the queue is searched from the beginning for an element where the timeout of the next element is greater than that of the new alarm. The alarm is then inserted into the queue before this element. If the new alarm is first in the queue the hardware alarm timer is

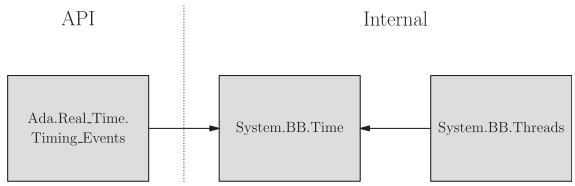

Fig. 2. Packages using alarm timers. An arrow pointing from one package to another indicates a dependency in the specification of the first (the other package is with'ed).

```
type Alarm_Descriptor is
   record
      Timeout : Time;
      --  Timeout of alarm or Time'First if alarm is not set

      Handler : Alarm_Handler;
      --  Handler to be called when the alarm expires or null if
      --  alarm is not set.

      Data : System.Address;
      --  Argument to be given when calling the handler

      Next, Prev : Alarm_Id;
      --  Next and previous elements when in alarm queue

   end record;
```

Listing 6. The private definition of the Alarm_Descriptor.

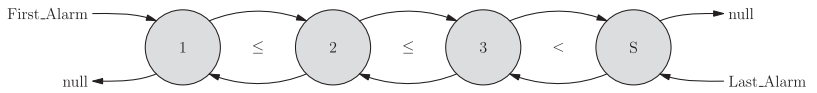

Fig. 3. Alarm queue with three alarms and sentinel at the end. Notice that the timeout value of the sentinel is always greater than that of the other alarms in the queue.

updated. The new alarm will always be inserted before the sentinel as the constant Time_Last exported through the real-time API is one less than the timeout Time'Last of the sentinel. This "trick" removes the need to check for the sentinel when searching the queue, which again speeds up the procedure.

When an alarm is cancelled using the procedure Cancel_Handler it is first checked whether the alarm is set or not. If the alarm is set it is extracted from the queue and cleared. If the extracted alarm was first in the queue the hardware alarm timer is updated. The procedure has no effect on alarms that are not set.

5.3. Interrupt handlers

There are two interrupt handlers used for timing services, the Clock_Handler called when the clock timer overflows and the Alarm_Wrapper called when the alarm timer expires. Both handlers have the highest interrupt priority and will therefore execute atomically, also with regard to each other, since all interrupts will be blocked while the handlers are executing.

The Clock_Handler increments the MSP of the system time and checks whether there is a pending alarm. An alarm may be pending due to computational delay when setting the alarm timer making it expire at the beginning of the next clock period. In this case no further action is taken by this handler, the pending alarm will be handled by Alarm_Wrapper. If there is no pending alarm and the timeout of First_Alarm is within a clock period the hardware alarm timer is set.

The Alarm_Wrapper is called when the alarm timer expires and is responsible for calling alarm handlers. The procedure first calls the handler of First_Alarm and then continues to the next alarm until all expired alarms are handled. Since alarm handlers may in turn call the procedures Set_Handler and Cancel_Handler the alarm queue has to be in a consistent state before calling them. In order to reduce overhead a flag signaling that the alarm timer should not be updated is set prior to calling the handlers. Thus the alarm timer is only updated after all handlers are executed. If the clock timer overflows while executing Alarm_Wrapper the MSP is updated and the clock interrupt cleared in order to reduce overhead. Due to this the alarm handler will be executed before the clock handler if both interrupt lines are asserted when the interrupt request register is read. This is done to reduce the overhead of alarm handlers.

5.4. Delaying and waking up threads

The Alarm_Time and Next_Alarm fields of the kernel type Thread_Descriptor were replaced by an Alarm_Descriptor. When a thread is to be delayed until the future time *T* the thread state is set to Delayed, the thread is extracted from the ready queue, and its alarm is set to expire at *T* with the handler Wakeup and the address of the thread as data. If *T* is not in the future the thread yields the processor by moving itself to the end of the ready queue for its priority.

The alarm handler wake-up is called when the alarm expires and typecasts the address argument to a thread pointer, sets its state to Runnable and inserts it into the ready queue. If necessary a context switch will be executed before returning to normal task execution.

5.5. Application programming interface (API)

The API is as defined in the Ada 2005 reference manual. The tagged type Timing_Event has an Alarm_Timer for setting off alarms and a Timing_Event_Handler for holding the current handler. The operations are performed atomically by using the kernel protection package to set and clear the global interrupt mask.

When a timing event is set with a handler its alarm is first cancelled using the procedure Cancel_Handler that removes a previously set alarm from the queue. If the alarm was not set this operation has little overhead. If the user handler is non-null the alarm is set with the handler Execute_Handler and the address of the timing event as data. The procedure Execute_Handler is called when the alarm expires and typecasts the address to a timing event access that is cleared before the handler is called. When a timing event is cancelled its alarm is first cancelled using the procedure Cancel_Handler and then Cancelled is set to true if the handler is non-null and false otherwise.

6. Implementation of execution time control

6.1. Design

The execution time control features were implemented in a new package System.BB.TMU where TMU stands for *Time Management Unit*. This packages implements all the functionality needed for execution time control and is used by several other packages as shown in Fig. 4.

The package System.BB.TMU defines the type CPU_Time representing execution time as a 64-bit modular integer, and the type Timer_Descriptor that is used for execution time control. The definition of the latter is shown in Listing 7. The access type Timer_Id points to timers and is used as argument for the operations of the package, following the same naming pattern as for threads and alarm timers.

The operations of the package System.BB.TMU may be divided into two categories: those used for managing timers, making sure that the correct timer is always active, and those performing operations on single timers. The first category of operations are called from internal packages in Fig. 4 that are not visible to the user, while the second category of operations are called by the user through the API. Both categories of operations need to call the TMU through wrappers in the package System.Tasking.Primitive_Operations if calling with an argument of the type Task_Id, as this has to be translated to the Thread_Id type that is used by the kernel. The operations of the TMU are detailed in the following sections.

There is a circular dependency in Fig. 4 between the TMU and threads packages caused by the first package needing to know about the type Thread_Id of the latter, while this package needs to know about Timer_Descriptor that is a component of the type Thread_Descriptor. The circular dependency is broken by using the new "limited with" construct of Ada 2005 which allows System.BB.TMU to have a limited view of the package System.BB.Threads. The type Threads_Id has to be redefined in the TMU package due to this limited view, but this is not a problem as it is possible to type-cast between the two types.

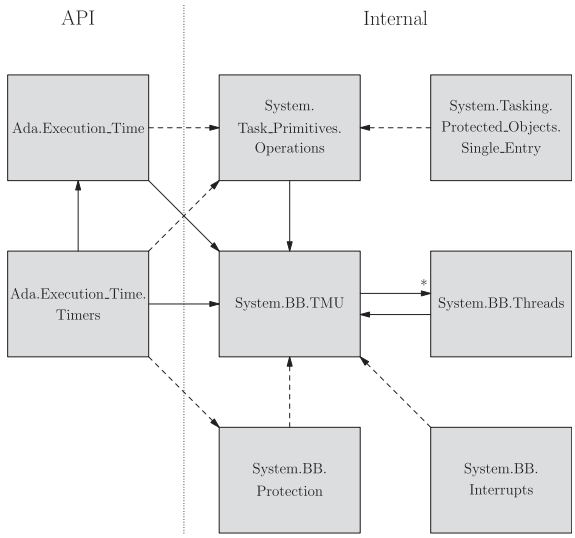

Fig. 4. Packages using the TMU. An arrow pointing from one package to another indicates a dependency in the specification of the first. The asterisk indicates the use of "limited with" for avoiding circular dependencies. The dashed arrow indicates that the package body is dependent on another package. All dependencies of the TMU package but the dependency of the threads package are omitted for simplicity.

```
type Timer_Descriptor is
   record
      Active_TM : Timer_Id;
      -- Will point to timer of another thread if this thread is
      -- executing code by proxy, otherwise to this timer.

      Base_Time : CPU_Time;
      pragma Volatile (Base_Time);
      -- Base time, updated when the timer is deactivated

      Timeout : CPU_Time;
      -- Timeout of timer or CPU_Time'First if timer is not set

      Handler : Timer_Handler;
      -- Handler to be called when the timer expires or null if
      -- timer not set.

      Data : System.Address;
      -- Argument to be given when calling handler

      Active : Boolean;
      -- Flag indicating if the timer is active (running)

      Acquired : Boolean;
      -- Flag indicating if the timer is acquired

   end record;
```

Listing 7. Private definition of timer type.

6.2. Timer management

There are several possible active timers in the context of the running task since there are separate timers for the pseudo tasks of system idling and interrupt handling, and the timer of another task is to be active when executing an entry by proxy. The pseudo task timers do not belong to real tasks and are declared in the body of the TMU package.

Immediately after system initialization there is one and only one active timer at any time. The first active timer is that of the environment task. After initialization the active timer is only changed as a result of one of the timer management operations being called. Normally the timer of the running task is active. The other possible timers in the context of a task and the relation among them are shown in Fig. 5.

Movement up- and downwards in Fig. 5 represents entering and leaving interrupt handlers. The corresponding operations of the TMU are Enter_Interrupt which takes the interrupt level as argument, and Leave_Interrupt that restores the previous active timer. These procedure are called from the interrupt handling system immediately after entering the low-level interrupt handler and just before leaving it. All interrupt levels but the highest

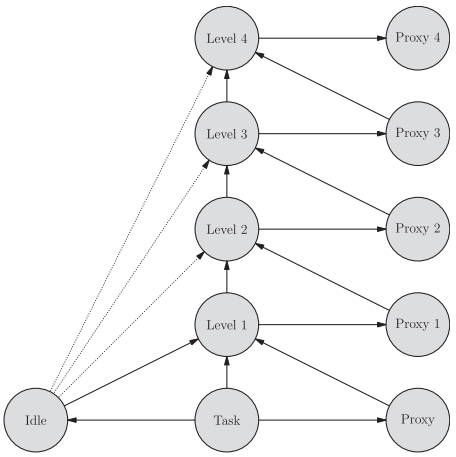

Fig. 5. Possible timers of a task. The figure is simplified by only showing an arrow representing interruption of one level by the next one. When traversing the graph the history is kept so that the correct timers may be restored later.

may be interrupted by a higher interrupt level. An entry by proxy may also be executed after the user provided interrupt handler has been executed. Context switches may occur before returning to normal execution, changing the running task and thus also the timer to be activated.

Movement from the center to right and back again represents using the timer of another task when executing an entry by proxy. The corresponding operations for this movement are Enter_Proxy that takes Thread_Id translated from the Task_Id of the task blocked on the entry as argument, and Leave_Proxy that restores the previous active timer.

Movement from the center to the left and back again is done when the task is executing the idle loop. The system has no dedicated idle task. Instead a task that has removed itself from the running queue when there are no other runnable tasks will insert itself at the head of the ready queue and enter the idle loop in the procedure Leave_Kernel of the package System.BB.Protection. The task does not execute any operations while idling but may of course be interrupted, otherwise no task would ever become runnable. At most one task will be idling at any time as the idling task is inserted into the ready queue. Prior to entering the idle loop the procedure Enter_Idle of the TMU is called to enable the idle timer. After leaving it Leave_Idle is called to restore the timer of the task that is now runnable. Tasks executing interrupt handlers will never enter the idle loop.

Internally a stack of timer accesses is used for keeping track of the different interrupt level timers. The bottom element of the stack always points to the timer of the running task and is updated when a context switch is executed. The stack is pushed when entering interrupt handlers moving upward in Fig. 5, and popped when leaving interrupt level handlers, moving downward in the same figure. The field Active_TM of Timer_Descriptor is used for keeping track of those computational states depicted as horizontal movement in Fig. 5, pointing to the timer of either the task blocked on the entry or the timer of the pseudo idle task after executing Enter_Proxy or Enter_Idle. The procedures Leave_Proxy and Leave_Idle restore the task on the top of the stack. Interrupt timers are pushed on the stack and activated by the procedure Enter_Interrupt while the procedure Leave_Interrupt pops the interrupt timer from the stack and reactivates Active_TM of the timer below on the stack.

6.3. Execution time measurement

The system measures the execution time as the number of CPU clock cycles used by a task since its activation. The Base_Time of a timer, also referred to as t_{base}, is initially zero and is used for keeping track of absolute execution time. The execution time for an active timer is the sum of the base time and the value of the COUNT system register, referred to as c:

$$t = t_{\text{base}} + c, \quad \text{where } c \in \{0, 1, \ldots, 2^{32} - 1\} \tag{2}$$

When a timer is deactivated its base time is updated:

$$t_{\text{base}} \leftarrow t_{\text{base}} + c \tag{3}$$

The Clock function returns the execution time of the given timer. If the timer is not active the function returns the base time of the timer as COUNT is only associated with the active timer. If active the function enters a loop where the base time and value of COUNT are read, the loop is exited if the base time has not been updated after being read. This allows the time to be read without blocking interrupts. After the loop the execution time is computed as the sum of the base time and COUNT value. There are three internal wrapper functions used for getting the clock of a thread timer, an interrupt level timer and the idle timer, named Thread_Clock, Interrupt_Clock and Idle_Clock respectively.

The execution time measurement depends on the CPU counter never overflowing. To prevent this the value C written to COMPARE is never greater than a constant C_{max} chosen to be $2^{31} - 1$. When COUNT equals this value a COMPARE interrupt will be pending as shown in Fig. 6, causing the timer to be inactivated and its base time updated when handled.

6.4. Execution time events

Execution time events for timers are supported by having a Timeout field also referred to as t_{timeout} specifying when the timer is to expire, an access to a procedure Handler that is to be called when the timer expires, and an address Data to be given as an argument when calling the handler. The procedure Set_Handler sets the handler to be called at the given time, the procedure Cancel_Handler cancels the handler, and the function Time_Remaining returns the CPU time remaining until the event. The COMPARE register is updated with the value C when timers are swapped, an active timer is set or an active and set timer is cancelled. If the timer is cleared C is C_{max} as before. If set, C is computed from $d = t_{\text{timeout}} - t_{\text{base}}$ by the following equation:

$$C = \begin{cases} 1 & \text{if } d < 1, \\ d & \text{if } 1 \leqslant d \text{ and } d \leqslant C_{max}, \\ C_{max} & \text{if } d > C_{max} \end{cases} \tag{4}$$

The procedure Compare_Handler is the handler for the COMPARE interrupt. The handler has the highest interrupt priority and the corresponding interrupt timer is therefore active when it is called. The timer causing the COMPARE interrupt has to be the timer pointed to by Active_TM of the timer below the top of the stack. The procedure gets this timer and first checks if it has expired since false interrupts may occur due to the overflow prevention earlier mentioned. If expired the timer is cleared and the handler called.

6.5. CPU counter primitives

Architecture dependent primitive operations written as inline assembler code were added to the package CPU_Primitives in order to manipulate the COUNT and COMPARE system registers. The function Get_Count returns the value of COUNT by using the

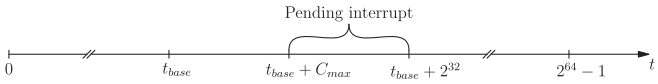

Fig. 6. Execution time for a timer from 0 to the final overflow.

system register read instruction. When inlined this function is a single instruction. The procedure Reset_Count takes the COMPARE value *C* as argument. In order to prevent a false interrupt COUNT is first set to *C*, then COMPARE to *C* and finally COUNT to 0. The procedure Swap_Count performs the same operation and returns the previous value of COUNT plus 4, which is the number of CPU cycles it takes to reprogram the COUNT register. This number of CPU cycles is added in order to avoid execution time leakage. The procedure Adjust_Compare changes the value of COMPARE to *C*. If the procedure is called with a *C* that is less than COUNT, the COMPARE interrupt will be pending after leaving the procedure.

6.6. Application programming interface

The execution time API is as specified by the Ada 2005 reference manual with the addition of the functions Interrupt_Clock and Idle_Clock shown in Listing 4 that call the corresponding functions of the TMU. The function Clock calls Thread_Clock of the TMU through a wrapper function that translates the Task_Id used by the API to the Thread_Id used by the kernel.

The execution time timers API is as specified in the reference manual with the addition of the tagged type Interrupt_Timer shown in Listing 5. The tagged type Timer includes a Timer_Id pointing to the associated kernel timer and a field holding the current handler. The Timer_Id is initialized the first time Set_Handler is called. The initialization procedure acquires the kernel timer for the interrupt priority corresponding to the discriminant if the timer is in Interrupt_Timer'Class or else the timer for the task ID. If the timer is already acquired or not available a Timer_Resource_Error is raised.

The operations on timer objects are performed in the same way as for timing events with the exception of timers not needing to be cancelled before they are set. As for timing events the package System.BB.Protection is used for executing the operations atomically.

7. Tests

7.1. Setup

The Atmel EVK1100 evaluation board with the UC3A0512 microcontroller [9] was used when testing the developed software. Test data was sent on the USART channel from the EVK1100 to an ordinary PC running GNU/Linux where it was retrieved and analyzed using GNU Octave [17]. The Atmel JTAG ICE Mk II was used for programming and debugging. The setup for the tests is shown in Fig. 7.

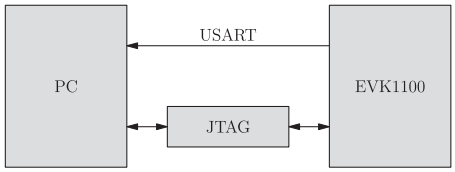

Fig. 7. Hardware setup for the tests.

A source of randomness was needed for the tests. For this purpose the pseudo-random simple multiplicative congruential algorithm was used to generate a uniform discrete distribution *X* between *A* and *B*:

$$s_0 = S \tag{5}$$

$$s_{i+1} = 7^5 \cdot s_i \bmod (2^{31} - 1) \tag{6}$$

$$x_i = A + \frac{(B - A + 1) \cdot s_i}{2^{31}} \tag{7}$$

With the chosen parameters this algorithm is referred to as the "Minimal Standard" and has been tested and used extensively over the years [18]. The algorithm has a period of 2,147,482,647 samples. The seed *S* is set when initializing the test program and should be the same every time so that the test results are reproducible.

7.2. Timing events

A test program for the timing event implementation was made as the Ada standard requires the following metric to be documented [1]:

> An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set.

We will refer to this metric as the *overhead* of calling the handler. The documentation requirement is somewhat unclear regarding the context for testing implementation, that is the number of timing events involved and if the effects of tasks and other interrupts are to be considered. These factors will affect the worst-case overhead, but may only be found on a per application basis. The documentation requirement is therefore interpreted to be for a *single* timing event with no running or ready tasks and no other interrupts than those internal to the system. For this implementation the only other interrupt will be the clock interrupt that occurs when the 16-bit hardware clock overruns. The criterion of goodness is the measured worst-case overhead, yet the best-case and average overhead will also be evaluated.

The program has one Timing_Event object, a protected object with the event handler, and uses the environment task to transfer test data to the PC. The test is run in batches, each starting with the event being set to occur at $t_e = t + (10000 + X)$ µs, where t is the current clock time and X is a pseudo-random distribution between 150 and 350 produced by the algorithm listed earlier. Batches terminate after 100 occurrences starting a new batch when the batch data has been transferred. The event handler reads the clock t, computes the overhead $D = t - t_e$, and stores this value together with X and the event time $t_e \bmod 2^{16}$ in the array holding the batch data. The latter of these values is used to find how close t_e is to the clock overflow interrupt which occurs when $t \bmod 2^{16} = 0$. A checksum is also computed and stored in the data array in order to verify the correctness of the data transferred to the PC. If the number of handler calls is less than 100 the event time is set to $t_e = t_e + X$ µs where X is distributed as earlier. Results based on five million recorded occurrences are shown in Table 1.

All observations with overhead of 22 or more, and 18 or less in Table 1 correspond to event occurrences with an offset in the range of −21–9 ticks relative to the clock overflow interrupt. The

Table 1
Observed overhead for timing event. The smallest observed overhead is 16 clock ticks or 8.53 μs, the largest is 30 ticks or 16 μs. The average overhead 10.6 μs. As seen from the table 99.98% of the observations are 19, 20 and 21.

Overhead		Observations	
Ticks	μs	Count	Freq. %
16	8.5333	30	0.0006
17	9.0667	67	0.00134
18	9.6000	85	0.0017
19	10.133	876764	17.535
20	10.667	3507692	70.154
21	11.200	614571	12.291
22	11.733	245	0.0049
23	12.267	80	0.0016
24	12.800	88	0.00176
25	13.333	82	0.00164
26	13.867	63	0.00126
27	14.400	67	0.00134
28	14.933	83	0.00166
29	15.467	73	0.00146
30	16.000	10	0.0002

overhead of all observations with offset in the range −25–15 are plotted in Fig. 8. All observations outside of this plot are either 19, 20 or 21.

7.3. Execution time drift

A test was made to check for drift of the execution time measurement against the system clock. If there is any drift accumulating over time the implementation fails the test.

The test program has four periodic tasks with characteristics shown in Table 2. The periodic tasks wake-up, busy wait for a given time C and delay until the next release. The system clock and the execution time of all tasks in the system, including the pseudo tasks and the environment task, are polled every major period (400 ms) by using a timing event. A background task sends these values to the PC where the difference D is computed as:

$$D = t_{\text{clock}} - \sum_{i \in \text{tasks}} t_{\text{execution time},i} \tag{8}$$

No drift in the execution time compared to the real-time clock was measured in 20000 samples gathered while running the test program. There was however small fluctuations in D caused by the CPU time measurement having 32 times higher resolution than the real-time clock.

Table 2
Periodic task set for execution time measurement.

Task	T (ms)	C (ms)	U (%)
A	50	5	10
B	100	10	10
C	200	20	10
D	400	40	10

7.4. Cost of interruption

There is some execution time cost of being interrupted since interrupt timers are not activated by hardware prior to entering interrupt handlers. A simple test program using only the environment task was made to measure this cost. The criterion of goodness is the worst-case cost to the interrupted task.

The program has a single loop where the execution time is first measured using Clock, then a busy wait for 50 ms (600,000 CPU cycles) is executed before the execution time is measured again. The two measured execution times are sent to the PC and the loop is repeated. Since the test program just has a single task only the system clock overflow interrupt calling the Clock_Handler is enabled. The 16-bit timer is clocked at 375 KHz and overflows every 174.76 ms, thus there will be at most one interruption between the clock calls. The difference between the shortest and longest measured execution time between the clock calls should give an indication on the overhead owing to interrupt preemption.

When running this test there were only three unique values of the difference between the clock before and after executing the busy wait: 600,050, 600,182 and 600,183. Thus the execution time is 600,050 when not interrupted, and 600,182 or 600,183 when interrupted, giving a worst measured interruption cost of 133 CPU clock cycles.

7.5. Cost of executing an entry by proxy

In the same way as with interrupts there is a non-zero execution time cost for a task executing an entry by proxy on behalf of another. A test program was made to measure this cost. The criterion of goodness is the worst-case cost to the task that executes the entry by proxy.

The test program has two tasks named *A* and *B* and a protected object. The protected object has an entry with a guard and a procedure that opens that guard, releasing any task blocked on the entry,

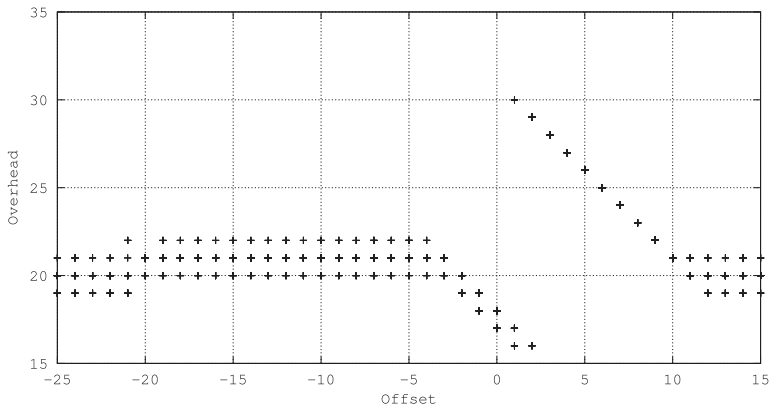

Fig. 8. Overhead of observations plotted against offset relative to clock overflow. The plotted values are in clock ticks and not μs. Notice that the best and worst-case overhead are for the same offset, depending on whether the alarm handler is called before the clock handler or not.

and has the highest system priority in order to avoid interruption while executing its operations. Task *A* has higher priority than *B* and measures its execution time before and after executing the protected procedure. After executing the procedure, *A* sends its execution time measurements to the PC before it is delayed until the next release. Task *B* will simply call the entry and then be delayed until the next release. The test is formed so that *A* will call the procedure both when *B* is blocked on the entry, in which case the entry will be executed by proxy, and when *B* is not blocked. This is done by releasing *A* sufficiently long (10 ms) after *B* half of the times.

When running the test the execution time for task *A* between reading the clock was found to be 387 CPU cycles when no entry by proxy was executed and 610 when the entry was executed, giving a cost of 223 CPU cycles. It should be noted that this is the best-case cost, the worst-case situation is when task *B* has a handler set at a time long into the future. This is due to the calculation of the COMPARE value *C* for task *B* when swapping timers. After modifying the test adding a timer of task *B* set to expire at CPU_Time'Last, the cost was then measured to be 237 CPU cycles.

8. Discussion

8.1. Timing events

The timing event features were fully integrated into the kernel timing services by adding an alarm mechanism. By using this mechanism both for timing events and waking up delayed tasks a clean implementation was achieved. Furthermore, lower overhead may be achieved as a timing event and a task release occurring at the same time are handled by the same alarm interrupt. This way, timing events may be piggybacked on task releases with very little cost. Internally, alarms are organized as an ordered queue with a sentinel at the end. The queue is doubly linked in order for alarms to be quickly extracted if cancelled, while the sentinel was added to simplify the code by removing the special case of having an empty queue. When inserting an alarm the queue is searched from the front to the end. The worst-case execution time is when the inserted alarm is to be before the sentinel in the queue, in which case all *N* alarms already present have to be searched. However the search loop is very tight, each iteration only takes seven CPU cycles.

The use of System.Address for passing data to handlers might be seen as a breach of Ada programming practice. One alternative would be to use a tagged type for passing data. This would be better from a design perspective, but would require a child package as the pragma No_Elaboration_Code applies to Threads and the use of tagged types requires elaboration code. Another possibility is to store user handlers in an array in Ada.Real_Time.Timing_Events and pass the array index instead of the address. This approach is taken by the GNATforLEON [7]. While this may be better from a design perspective it requires additional memory and puts an artificial limit on the number of timing event objects possible. The use of System.Address was chosen for efficiency, and considered safe as the handler and data are always set and cleared at the same time, reducing the risk of calling a handler with a wrong argument. The same considerations apply for the implementation of execution time timers.

8.1.1. Overhead

The Ada 2005 standard [1] does not specify how the overhead for timing events should be measured. The worst-case overhead will be greater for systems where many timing events may occur at once than those with only one timing event. If many timing events the occur at once the execution time for each event handler will also affect the worst-case overhead. Furthermore tasks and other interrupts also affect the overhead as all interrupts are blocked when executing within the kernel. These factors will vary between applications, so a measurement of the overhead for one application will have limited value for another. In order to have meaningful results the documentation requirement was interpreted to be for a minimal system having only a single timing event that is not affected by other factors than those internal to the kernel. In the authors opinion the Ada standard should be revised to be more specific on conditions for this documentation requirement.

The test program used has a single timing event, no tasks that can interfere with this event, and no other interrupts enabled than those used internally in the kernel. Thus only the clock interrupt may cause timing events to be delayed. As seen from the test results in Table 1 the worst-case measured overhead is fairly low, only 16 μs or 960 CPU cycles. The average overhead is 10.64 μs or about 638 CPU cycles. As expected the worst-case overhead occurs when timing events are set close to the clock overflow, so that the clock handler is called before the alarm handler. This situation can be seen at offset 1 in Fig. 8. Curiously the best-case also occurs at this offset. The reason is that if the clock interrupt is asserted and interrupts the processor, and the alarm interrupt is asserted just before the interrupt request register is read by the low-level interrupt handler, then the alarm handler is called instead and will have saved some time. The alarm timer is set with a value relative to the clock, so there is a lag of a few ticks between the hardware clock and the alarm clock. The overhead values of 22 in the offset range −21 to −4 are caused by the clock overflowing before the clock is read. This causes some extra computation in the function Clock called from the event handler, adding to the overhead.

8.2. Execution time control

The timer management is the main difference between the design of execution time features in this work and others. While other implementations such as GNATforLEON version 1.3 [7] only measure the execution time of the running task between context switches, this implementation improves accuracy by taking into account the effects of interrupts and execution by proxy.

Accuracy of execution time measurement for tasks is improved by charging the execution time of interrupt handler to a pseudo interrupt task of the corresponding interrupt level. This allows task budgets to be tighter and therefore higher utilization of the processor. Furthermore, it is possible to use execution time timers on interrupt levels in order to set a budget and block interrupts if this budget is exceeded. Timers are not used for the highest interrupt level since the kernel interrupts are of this level. Blocking these interrupts will result in system malfunction. The cost of interruption by the clock interrupt, found by testing to be 133 CPU cycles, is the best-case cost for general interrupts as the highest level interrupt timer is never set. The additional cost of being interrupted by an interrupt timer with a handler set far into the future is 14 CPU cycles as observed with execution of entries by proxy, giving a worst-case cost of 147 CPU cycles.

Execution by proxy improves system performance by reducing the number of context switches needed. When task *A* executes a protected operation releasing task *B* that is blocked on an entry, *A* will also execute the entry on behalf of *B*. In this work *B* is charged the execution time spent on the entry thereby improving the accuracy of the execution time measurement as this execution time does not have to be added to the budget of *A*. The usefulness of this feature is however highly dependent of the implementation overhead of changing timers compared to the execution time of the entry. As seen from the test results the worst-case cost of executing an entry by proxy is 237 CPU cycles, which definitely is more than

the cost of executing an entry with a null body. Still the benefit of having a known constant worst-case cost instead of an unknown cost when executing entries by proxy seems to justify the overhead.

The implementation also measures the time spent on system idling. One potential use of this feature is to read the execution time spent on idling periodically and use this as a measurement of CPU utilization within the time interval.

8.3. Execution time measurement

The correctness of the execution time measurement relies on the COUNT register never overflowing. The use of a maximal value $C_{max} = 2^{31} - 1$ for COMPARE guarantees that the COUNT register will never overflow as long as the system does not run with interrupts disabled for more than 2^{31} clock cycles, which is about 179 s when clocked at 12 MHz. We find it reasonable to assume that systems tasks and interrupts will not be continuously active long enough for c to equal C_{max}. Typically real-time systems will have several context switches and interrupts each second making them safe from overflows. Therefore one could argue that the use of C_{max} is unneeded altogether, yet this additional safety comes at a low cost and may also be needed for systems with very high CPU frequency.

The test for execution time drift gave conclusive results that there are no leakage of execution time as no accumulative drift was observed over a long period of time. The only place leakage could occur is in the procedure Swap_Count of the CPU primitives due to an incorrect number of cycles added to the previous COUNT value. The number of cycles needed to complete these instructions will always be the same since interrupts are disabled and there are no branches or memory accesses between reading the previous value and writing the new one.

8.4. Portability and hardware support

The implementation of the proposed interrupt execution time control features were done on the AVR32 UC3 microcontroller. Unfortunately recent changes to the COUNT/COMPARE semantics [9] for the UC3 core has made the registers less useful for execution timing. The new semantics state that when COUNT equals COMPARE the interrupt line should be asserted and COUNT set to 0. The original semantics were used in this paper since only the engineering presample version was available with the EVK1100 at the time of development. It is also assumed that new hardware with semantics similar to the original CPU counter semantics will be introduced in later revisions of the UC3.

It is possible to implement the execution time control features using the new semantics, however this would require additional code to prevent the COUNT register from equaling COMPARE and being reset several times before the COMPARE interrupt is handled resulting in execution time drift. Such a situation would occur if COMPARE has a low value. The situation could be avoided by adding a sufficiently large offset value to both COUNT and COMPARE, and adding code to handle this offset. All of this could be done within in the CPU primitives, requiring no change to the TMU. However the change would increase the overhead associated with the execution time control features.

The bare-board GNAT implementation of the Ada 2005 execution time features is portable to other architectures as long as it is possible to implement the CPU primitives with fairly low overhead. The features should also be portable to other embedded run-time systems that allow direct control of the timer hardware. In order to port the features to operating systems such as Linux, the kernel would have to be modified to support interrupt execution time measurement. This would be harder, but is fully possible.

8.5. Applications

The Ravenscar profile forbids the use of asynchronous task control and requires the priority of tasks to be constant. Therefore an execution time overrun handler can not terminate the task that exceeded its budget or reduce its priority as is possible for programs using the full Ada 2005 tasking model [5]. An alternative action the handler can take is to set a timeout-flag that is polled by the task and a fallback handler [19]. The next time the offending task polls the timeout-flag it will see that the flag is set, cancel the fallback handler and voluntarily give up control over the processor. If the task fails to cancel the fallback handler it should be treated as a system error.

The Deferrable Server algorithm for single tasks may be implemented with the functionality provided by the execution time control and timing event features. A timing event may be used to reset the execution time budget periodically. Overruns have to be handled by setting a flag and a fallback handler as described earlier. The Deferrable Server algorithm may also be used for interrupt handling, disabling the interrupt upon execution time overrun and re-enabling it when the execution time budget of the interrupt level is replenished [19]. A fallback handler may be used to ensure system safety. The use of the Deferrable Server algorithm for interrupt handling makes it possible to protect the system against bursts of interrupts that would otherwise result in system failure.

The flag polling combined with the fallback handler provides the safety needed to prevent overruns of high priority tasks making lower priority tasks miss their deadlines, yet it is neither a very elegant solution nor an efficient one. Other solutions should be evaluated, possibly defining an extended Ravenscar profile allowing limited asynchronous control or dual-band scheduling. This would allow more efficient and elegant handling of execution time overruns for tasks.

8.6. Other programming languages

The implementation described in this paper is written in Ada 2005 for the GNAT run-time library. However the concepts presented in this paper are not limited to Ada 2005, and may be implemented in other programming languages and real-time systems.

8.6.1. POSIX

The real-time extension to POSIX defines execution time control for POSIX processes and threads [20]. The POSIX specification uses the type clockid_t to identify clocks such as the real-time clock and execution time clocks, and the type timer_id to identify timers. The general model is the same as in this paper, each timer is associated with a clock, and a clock may have several timers. The POSIX functions used for clocks and timers shown in Listing 8.

The functions defined in Listing 8 provide similar features for execution time control as those defined for Ada 2005. The function clock_gettime retrieves the current time of a clock, while the function timer_create is used to create a timer associated with a given clock. Note that in POSIX a timer can only be used to generate one type of signal event, specified by the argument of type sigevent provided at creation. After a timer is created it may be set to expire in a given time using timer_settime. An armed timer is cancelled using the same function with a zero time argument. The function timer_gettime is used to get the time until the event occurs. Only one signal may be pending for a timer at any time, if more events occur before it is handled the timer overruns. If the real-time signals extension is supported the function timer_getoverrun returns the number of overruns for a timer.

The POSIX definition of execution time control is quite similar to that of Ada 2005. This is no surprise as the Ada features were designed to be implemented using POSIX timers [6]. Due to the

```
int clock_gettime(clockid_t, struct timespec *);
...
int timer_create(clockid_t, struct sigevent *restrict, timer_t *restrict);
int timer_delete(timer_t);
int timer_gettime(timer_t, struct itimerspec *);
int timer_getoverrun(timer_t);
int timer_settime(timer_t, int, const struct itimerspec *restrict,
        struct itimerspec *restrict);
```

Listing 8. POSIX clocks and timers defined in "`time.h`".

similarity the implementation described in this paper could serve as a blueprint for POSIX implementations. No changes to the POSIX API would be needed as the implementation would only have to provide a system specific clockid_t for each interrupt level in order to support interrupt execution time control.

8.6.2. RTSJ

The Real-Time Specification for Java (RTSJ) [21] takes an integrated approach to execution time control, referred to by the specification as *cost monitoring*. In essence the RTSJ allows budgets, or the *cost*, to be set for periodic threads [22]. The cost is the same for each release and is set by calling for instance the method setReleaseParametersIfFeasible of a RealTimeThread object. Periodic release of the thread object is initiated by calling schedulePeriodic. In the case of a cost overrun the offending thread will only be allowed to continue executing if this will not cause lower priority thread to miss their deadlines. Otherwise the thread will be immediately blocked until its next release. The cost monitoring scheme is intended to be independent of the scheduling policy [22].

Clocks and timers as described in this paper may be used "under the hood" to implement cost monitoring in RTSJ run-time environments. Furthermore RTSJ applications applying cost monitoring would also benefit from the improved accuracy provided by not charging the running thread the execution time of interrupt handlers. However, since interrupts are asynchronous events they cannot be integrated into the current model that only supports cost monitoring for periodic threads. Santos and Wellings have pointed to errors in the current model and made suggestion for a new improved model for cost monitoring [22]. Their proposed model corrects these errors and also allows cost monitoring for sporadic and aperiodic schedulable objects such as asynchronous event handlers. If their proposal is included in the RTSJ, then the ideas of this paper could be applied to allow cost monitoring for interrupt handling for Java.

9. Conclusion

The main contribution of this work is an implementation of the new Ada 2005 execution time control features that addresses inaccuracies reported by other implementations [7]. Our implementation does not charge the execution time of interrupt handlers on the interrupted task, this time is instead charged a pseudo interrupt task for the given interrupt priority. Furthermore, a task executing an entry by proxy on behalf of another task is not charged the execution time of the entry, this time is instead charged the task blocked on the entry. The benefits of this are twofold:

1. The accuracy for task execution time measurement is improved, allowing task budgets to be tighter, and thereby allowing higher CPU utilization.
2. It is possible to monitor and control the execution time spent on handling interrupts of a given priority using execution time servers. This makes it possible to protect the systems from bursts of interrupts that could otherwise result in tasks missing their deadline.

The increased accuracy offered by this implementation makes it possible to set tight execution time budgets for tasks as the worst-case execution time cost of interruption and executing an entry by proxy are reduced to known constants. This will make development of real-time applications easier and allow higher CPU utilization without compromising safety. The addition of execution time clocks and timers for each interrupt priority pseudo task also allows measurement and restriction on the time spent on interrupt handling by the use of such methods as the Deferrable Server algorithm (new interrupt occurrences may be blocked even with the Ravenscar profile). This increases the safety of applications as burst of interrupts that may have caused system failure now can be handled in a simple and efficient manner.

The new Ada 2005 timing events feature were also implemented. A flexible and efficient alarm mechanism was added to the timing services of the kernel supporting both timing events and task wake-up. Since they use the same alarm mechanism timing events may be piggybacked on task releases with very low additional cost since the same interrupt handler call will serve both operations.

The Ravenscar profile does not support asynchronous control or dynamic task priorities, therefore applications for now have to resort to methods such as flag polling for detecting overflows and voluntarily giving away control over the processor. Other possible usages is just to log the overruns and only take action if they exceed a certain number. In order to allow more efficient execution time control an extension to the Ravenscar profile allowing execution time timers, limited asynchronous control and dual-band priorities should be developed and evaluated.

Although this implementation of the new Ada 2005 real-time features is for the AVR32 it can easily be ported to any architecture having similar timer functionality as the AVR32. Furthermore the usefulness of execution time control for interrupts goes beyond the Ada programming language. In the authors opinion the design presented in this paper should be applicable and useful for a wide range of programming languages and computer architectures.

Acknowledgment

Thanks to professor Alan Burns for giving valuable advice on the Ada 2005 execution time control features and documentation requirements for timing events. We would also like to express our gratitude to Atmel Norway for advice and providing development tools free of charge.

References

[1] ISO/IEC, Ada Reference Manual – ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1. URL: <http://www.adaic.com/standards/05rm/html/RM-TOC.html>.
[2] J. Barnes, Rationale for Ada 2005, John Barnes Informatics, 2007.
[3] A. Wellings, A. Burns, Ada-Europe 2007, Springer Berlin/Heidelberg, 2007, Ch. Real-Time Utilities for Ada 2005, pp. 1–14. doi:10.1007/978-3-540-73230-3.
[4] R. Wilhelm et al., The worst-case execution-time problem – overview of methods and survey of tools, Trans. Embedded Comput. Syst. 7 (3) (2008) 1–53. doi:http://doi.acm.org/10.1145/1347375.1347389.

[5] A. Burns, A. Wellings, Programming execution-time servers in Ada 2005, in: Proceedings on 27th IEEE International Real-Time Systems Symposium RTSS '06, 2006, pp. 47–56. doi:10.1109/RTSS.2006.39.
[6] M.G. Harbour, et al., Implementing and using execution time clocks in Ada hard real-time applications, in: Lecture Notes in Computer Science, vol. 1411/1998, Springer, Berlin/Heidelberg, 1998, pp. 90–101.
[7] S. Urueña, J. Pulido, J. Redondo, J. Zamorano, Implementing the new Ada 2005 real-time features on a bare board kernel, Ada Lett. XXVII (2) (2007) 61–66. doi:http://doi.acm.org/10.1145/1316002.1316016.
[8] A. Wellings, Implementation experience with Ada 2005, Ada Lett. XXVII 2 (2007) 59–60. session report..
[9] Atmel Corporation, AT32UC3A Series – Preliminary Datasheet, November 2009. URL: <http://atmel.com/dyn/resources/prod_documents/doc32058.pdf>.
[10] Atmel Corporation, AVR32 – Architecture Document, November 2007. URL: <http://atmel.com/dyn/resources/prod_documents/doc32000.pdf>.
[11] K. Gregertsen, A. Skavhaug, An efficient and deterministic multi-tasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller, in: Design, Automation and Test in Europe Conference and Exhibition, 2009, pp. 1572–1575.
[12] J.F. Ruiz, GNAT pro for on-board misson-critical space applications, Ada-Europe. URL: <http://www.adacore.com/wp-content/uploads/2005/05/GNAT_Space_Apps.pdf>.
[13] A. Burns, The Ravenscar profile, Ada Lett. XIX (4) (1999) 49–52. doi:http://doi.acm.org/10.1145/340396.340450.
[14] A. Burns, B. Dobbing, T. Vardanega, Guide for the use of the Ada Ravenscar profile in high integrity systems, Ada Lett. XXIV (2) (2004) 1–74. doi:http://doi.acm.org/10.1145/997119.997120.
[15] J.A. de la Puente, J. Zamorano, Execution-time clocks and Ravenscar kernels, Ada Lett. XXIII (4) (2003) 82–86. doi:http://doi.acm.org/10.1145/959221.959237.
[16] J.A. de la Puente, J. Zamorano, J. Ruiz, R. Fernández, R. García, The design and implementation of the Open Ravenscar Kernel, in: IRTAW '00: Proceedings of the 10th International Workshop on Real-time Ada, ACM, New York, NY, USA, 2001, pp. 85–90. doi:http://doi.acm.org/10.1145/374370.374387.
[17] J.W. Eaton, GNU Octave: Interactive language for numerical computations, GNU. URL <http://www.gnu.org/software/octave/doc/interpreter/>.
[18] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recepies in C++, Cambridge University Press, 2002.
[19] K.N. Gregertsen, A. Skavhaug, A real-time framework for Ada 2005 and the Ravenscar profile, in: 35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '09, 2009, pp. 515–522. doi:10.1109/SEAA.2009.40.
[20] The Open Group base specifications issue 6, IEEE Std 1003.1, <http://www.opengroup.org/onlinepubs/000095399/>.
[21] RTSJ – The Real-time Specification for Java. <http://www.rtsj.org/>.
[22] O.M. Santos, A. Wellings, Cost enforcement in the real-time specification for Java, Real-Time Syst. 37 (2) (2007) 139–179.

Kristoffer Nyborg Gregertsen received his M.Sc. degree in engineering cybernetics from the Norwegian University of Science and Technology (NTNU) in 2008. He is currently taking his Ph.D. in engineering cybernetics at NTNU. The topic of his Ph.D. is software and hardware support for execution time monitoring in real-time systems. He has also worked for Atmel Norway developing device drivers for AVR32 Linux.

Amund Skavhaug received his siv.ing. (M.Sc.) degree in engineering cybernetics from the Norwegian Institute of Technology (NTH) in 1991, and his doctoral degree in Cybernetics from NTNU in 1997. He has been working as a researcher in several companies, including SINTEF and Statoil. He co-chairs the Dependable Embedded Systems work-group in ERCIM and is a board member and Secretary General for EUROMICRO. He is currently an associate professor at NTNU, where his main interests currently is utilization of complex embedded systems and enabling technologies for their development.

A.5 Article No. 5

K. N. Gregertsen and A. Skavhaug:

“Functional specification of a Time Management Unit”

Presented at SAFECOMP2010 / DECOS workshop [24].

Functional specification for a Time Management Unit

Kristoffer Nyborg Gregertsen, Amund Skavhaug
Department of Engineering Cybernetics, NTNU
N-7491 Trondheim, Norway
{*gregerts,amund*}*@itk.ntnu.no*

Abstract

This paper gives a functional specification for a Time Management Unit (TMU) used to support execution time control in real-time systems. The TMU is designed for efficient access though the high-speed bus (HSB) of the microcontroller. The simplicity of the described TMU allows it to be added to existing System-on-Chip (SoC) designs with minimal effort.

1. Introduction

Real-time scheduling algorithms usually depend on the worst-case execution time (WCET) of tasks being known in order to guarantee for all deadlines being reached. For modern computer architectures using performance enhancing techniques such as pipelines, cache and branch prediction, finding this WCET may be very hard [1], and therefore prohibitively expensive and time consuming for most embedded projects. Also the WCET will often be much greater than the average execution time. Given that the WCET is not known the developer has to choose whether to use conservative budgets and have poor CPU utilization, or to use optimistic budgets and risk deadlines occasionally being lost. Neither of the alternatives may be acceptable.

By using execution time control developers may use less conservative budgets and handle overruns dynamically in order to prevent deadline loss. In order to do this the run-time system has to provide execution time clocks that measure the total time an executable entity has been running on the system. The clock is started when the entity is scheduled for execution and stopped when it is done executing or preempted by another entity. A timer is associated with a clock and is used to call an event handler when its clock reaches a specified time. We will refer to clocks and timers combined as execution time monitoring.

Execution time monitoring for Ada 2005 was implemented on the AVR32 architecture by the authors at NTNU [2]. Our implementation differs from other implementations known to the authors by not charging the interrupted task the execution time of interrupt handlers. This time was instead charged a pseudo interrupt task for the given interrupt priority. The benefits of doing this are twofold: (1) The accuracy for task execution time measurement is improved, allowing tighter task budgets, and thereby higher CPU utilization. (2) It is possible to monitor and control the execution time spent on handling interrupts of a given level. This makes it possible to protect the systems from bursts of interrupts that could otherwise result in tasks missing their deadline.

For these features to be efficient the overhead of switching clocks when entering an interrupt handler and performing a context switch should be as low as possible, preferably also with a deterministic execution time. The AVR32 implementation of execution time monitoring used the CPU cycle COUNT / COMPARE system registers of the architecture. Since these registers are only of 32-bits, relative time had to be used resulting in computational overhead when translating the absolute execution time into a relative one. Special care was also needed to prevent COUNT from overflowing. The lack of timers that allow sufficient efficiency for execution time monitoring motivates a dedicated timer unit which we will refer to as a Time Management Unit (TMU).

A TMU for the LEON architecture has been designed at NTNU and implemented on a FPGA [3]. This design is more complex than the one described in this paper. This TMU has the IRQ lines as input from the interrupt controller and forwards these to the core. The active timer is changed according to the run-level. Therefore it has one set of COUNT / COMPARE registers for the ordinary execution level and each interrupt level. If the budget for an interrupt level is exhausted it is masked, and the CPU would not handle interrupts of this level until the budget is replenished. The replenishing of interrupt execution time budgets is

also designed to be done within the TMU, using either the sporadic or deferrable server algorithms. While this TMU is design powerful it considered too complex to implement and not flexible enough to be useful for different real-time systems.

In the following a functional description of a simple yet efficient TMU is given using the SystemC modeling library [4]. Then follows examples of how to use the TMU to implement execution time monitoring on the AVR32 architecture [5]. Finally the design of the TMU is discussed, and a conclusion and plan for a hardware implementation on the AVR32 UC3 microcontroller series [6] are given.

2. Functional specification

The TMU is designed as a memory-mapped slave device accessible by the processor core though the high-speed bus (HSB) as shown in Figure 1. The HSB address and data are both assumed to be 32-bit wide. The registers available on the high-speed bus are listed in Table 1.

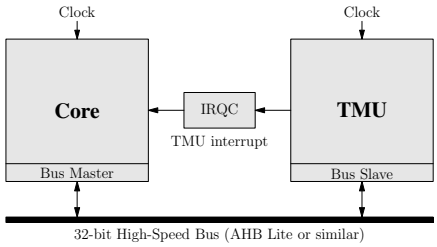

Figure 1. Core and TMU connected to the HSB.

Table 1. Memory interface of the TMU.

Offset	Register	Reset state
0x0	TMU_COMPARE_HI	0xffffffff
0x4	TMU_COMPARE_LO	0xffffffff
0x8	TMU_COUNT_HI	0
0xc	TMU_COUNT_LO	0
0x10	TMU_SWAP_COMPARE_HI	0xffffffff
0x14	TMU_SWAP_COMPARE_LO	0xffffffff
0x18	TMU_SWAP_COUNT_HI	0
0x1c	TMU_SWAP_COUNT_LO	0

In addition to HSB interface the TMU has a clock signal as input. This clock need not be the same as that used by the core. The TMU also generates an interrupt signal. This signal will usually be routed to the core through the interrupt controller as shown in Figure 1. The SystemC definition of the TMU module is shown below:

```
class tmu : public sc_module, public hsb_if
{
public:
  sc_in<bool>  clock;
  sc_out<bool> interrupt;

  bool read(hsb_addr addr, hsb_data& data);
  bool write(hsb_addr addr, hsb_data data);

  tmu(sc_module_name name_, hsb_addr base)
  : sc_module(name_)
  {
    SC_HAS_PROCESS(tmu);
    SC_CTHREAD(tick, clock);
    this->base = base;

    compare = swap[0] = -1;
    count   = swap[1] = 0;
  }

private:
  void tick();
  sc_uint<64> compare, count, swap[2];
  sc_uint<32> buffer;
  hsb_addr base;
};
```

Internally the TMU has a 64-bit COUNT register that is incremented on every positive edge of the clock signal. After COUNT is incremented it is compared with the 64-bit COMPARE register. If COUNT is greater than or equal to COMPARE then the interrupt signal is asserted. The SystemC code for incrementing COUNT is shown below:

```
void tmu::tick()
{
  interrupt = false;

  while (true) {

    // Wait for positive clock flank
    wait();

    // Increment count and check for overrun
    count++;
    interrupt = (compare <= count);
  }
}
```

In order to atomically swap a new set of COUNT / COMPARE values with the current two 64-bit swap registers are provided. The swap operation is performed when the low-word of SWAP_COUNT is written. After the operation the previous values of COUNT and COMPARE may be read from the swap registers.

The COUNT and COMPARE registers may also be accessed directly by using an internal 32-bit buffer. When reading the high-word of COUNT or COMPARE the low-word is buffered, and this buffered value is returned when subsequently reading the low-word. When writing the high-word of COUNT or COMPARE the high-word value is buffered. When the low-word subsequently is written the register is updated using the buffered high-word and the provided low-word value. A shortened version of the SystemC code for accessing the TMU registers is shown below:

```
bool tmu::read(hsb_addr addr, hsb_data& data)
{
  switch (addr - base) {

  case TMU_COMPARE_HI:
    // Return high-word and store low-word in buffer
    data = compare.range(63,32);
    buffer = compare.range(31,0);
    break;

  case TMU_COMPARE_LO:
    // Return low-word from buffer
    data = buffer;
    break;

  ...

  case TMU_SWAP_COUNT_HI:
    // Return high-word of SWAP[1]
    data = swap[1].range(63,32);
    break;

  case TMU_SWAP_COUNT_LO:
    // Return low-word of SWAP[1]
    data = swap[1].range(31,0);
    break;

  default:
    return false;
  }
  return true;
}

bool tmu::write(hsb_addr addr, hsb_data data)
{
  switch (addr - base) {

  case TMU_COMPARE_HI:
    // Store high-word in buffer
    buffer = data;
    break;

  case TMU_COMPARE_LO:
    // Update COMPARE
    compare.range(63,32) = buffer;
    compare.range(31,0) = data;
    break;

  ...

  case TMU_SWAP_COUNT_HI:
    // Update SWAP[1] high-word
    swap[1].range(63,32) = data;
    break;

  case TMU_SWAP_COUNT_LO:
    // Update SWAP[1] low-word
    swap[1].range(31,0) = data;

    // Swap COMPARE and COUNT
    {
      sc_uint<64> tmp[2] = {compare, count};
      compare = swap[0];
      count   = swap[1];

      swap[0] = tmp[0];
      swap[1] = tmp[1];
    }
    break;

  default:
    return false;
  };
  return true;
}
```

Care must be taken not to interleave the writing of the high and low-word of COUNT and COMPARE with other accesses as this could cause unwanted behavior. It is recommended to use 64-bit load / store instructions such as those available for the AVR32 architecture so that both the registers are read or written in one atomic operation.

3. Usage

The design of the TMU and the powerful load / store instructions of the AVR32 architecture allows efficient operations such as writing the whole TMU context using one instruction. In this code the COUNT and COMPARE register values are stored in memory together with the task context next after the register values. Thus this implementation supports at most one timer for each execution time clock. The context switch may then be done as follows:

```
/* Store CPU-context of running thread */
lda.w    r8, running_thread
ld.w     r9, r8[0]
sub      r9, -60
stm      --r9, r0-r7,sp,lr
mfsr     r0, SYSREG_SR
st.w     --r9, r0

/* Load address of first thread */
lda.w    r10, first_thread
ld.w     r10, r10[0]
st.w     r8[0], r10

/* Load TMU-context of first thread */
lda.w    r8, tmu_address
sub      r8, -16
ldm      r10++, r4-r7
stm      r8++, r4-r7

/* Store TMU-context of running thread */
sub      r8, 16
ldm      r8++, r4-r7
stm      --r9, r4-r7

/* Load CPU-context of first thread */
ld.w     r0, r10++
mtsr     SYSREG_SR, r0
sub      pc, -2
ldm      r10++, r0-r7,sp,pc
```

Changing to an interrupt clock after entering the low-level interrupt handler may be done in the same way as for the context switch. Reading the clock is just a matter of reading the double-word stored at TMU_COUNT and may be done directly from high level code. To set or adjust a timer one only has to write the double-word at TMU_COMPARE. If the written value is less than COUNT the interrupt line will be asserted in the next clock cycle.

It should also be noted that if only execution time measurement is to be used it suffices to write to TMU_COUNT and TMU_SWAP_COUNT. The COMPARE value will then remain in the initial state.

4. Discussion

While the earlier AVR32 implementation of execution time monitoring using the COUNT / COMPARE system registers works according to its specification, it inflicts a too high overhead on interrupt handling and context switches, as software logic is needed to use the smaller 32-bit registers for the 64-bit execution time values used by applications [2]. The WCET for switching execution time clocks for this implementation is about 50 CPU cycles, adding about 100 cycles to the overhead of each task interruption and context switch. The overhead varies depending on whether the timer is set to expire within the next 2^{31} clock cycles or not. By adding execution time monitoring the WCET of the context switch more than doubles.

This experience was used for the TMU specification presented in this paper. By using 64-bit registers for COUNT and COMPARE there is no need for translating the absolute execution time values used by applications into relative values loaded into the registers. Also by having a less-or-equal comparison instead of only equality it is no longer necessary to check if COMPARE is less than COUNT before setting the registers. Most important however, is the ability of the TMU to swap COUNT / COMPARE registers atomically. As seen by the context switch code example this makes it possible to change the context of the TMU using only two instructions. Each of these instructions will only take the time needed to transfer 4 words over the bus, and the execution time of the context switch will be the same regardless of whether or when the timers are set given that there are no other masters with higher priority accessing the bus. If the system has several cores, as some embedded architectures now have, then one TMU module is needed for each core.

The TMU of this paper is much simpler than that designed and implemented for the LEON architecture [3]. This more complex TMU takes IRQ lines from the interrupt controller as inputs and changes execution time clocks automatically when the processor is to enter an interrupt level. It also masks the handling of an interrupt level according to a predefined policy. This makes it hard to get this design implemented on existing SoCs as the IRQ signals a vital part of the system that microcontroller providers will probably be reluctant to change. One benefit of the complex TMU is that it has zero overhead on interrupt handling. This comes at the cost of low flexibility, as the execution time control policy is hard-coded. In contrast the simple TMU of this paper does not modify interrupt signals and leaves policy entirely to software, allowing different run-time systems and applications to user their own policies. This simplicity makes this TMU easier to implement on different hardware architectures and useful for a wider range of real-time systems.

5. Conclusion

The main contribution of this paper is the functional specification of a simple and efficient Time Management Unit (TMU) for facilitating high-resolution execution time control on real-time systems. The TMU is designed for maximal flexibility leaving the policy decisions of the execution-time control entirely to the software. By using special swap registers the TMU allows for efficient atomic change of the running execution time clock for instance when entering an interrupt level or performing a context switch. The TMU is accessible as a memory-mapped slave through the high-speed bus and may thus be used for a wide range of existing System-on-Chip (SoC) designs.

6. Further work

Work is being initiated at NTNU to implement the described TMU on the UC3 microcontroller series of the Atmel AVR32 architecture in close cooperation with Atmel Norway. The TMU will be used for the AVR32 implementation of the Ada 2005 execution-time control features.

References

[1] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[2] K. N. Gregertsen and A. Skavhaug, "Implementing the new Ada 2005 timing event and execution-time control features on the AVR32 architecture," February 2009, submitted to Journal of Systems Architecture.

[3] B. Forsman, "A Time Management Unit (TMU) for real-time systems," Master's thesis, Norwegian University of Science and Technology (NTNU), 2008.

[4] *Standard SystemC Language Reference Manual*, IEEE Std. 1666–2005.

[5] Atmel Corporation, *AVR32 - Architecture Document*, November 2007. [Online]. Available: http://atmel.com/dyn/resources/prod_documents/doc32000.pdf

[6] ——, *AVR32UC3 - Technical Reference Manual*, March 2010. [Online]. Available: http://atmel.com/dyn/resources/prod_documents/doc32002.pdf

A.6 Article No. 6

K. N. Gregertsen and A. Skavhaug:

“Implementation and usage of the new Ada 2012 execution time control features”

Published in Ada User Journal [31].

Implementation and Usage of the new Ada 2012 Execution Time Control Features

Kristoffer Nyborg Gregertsen, Amund Skavhaug
Department of Engineering Cybernetics, NTNU, Trondheim, Norway; email: {gregerts,amund}@itk.ntnu.no

Abstract

This paper describes an implementation of Ada 2012 execution time control supporting the new separate execution time clocks for interrupts that has a design with several benefits. The real-time and execution time features use the same clock and alarm abstraction reducing the amount of code needed for the implementation. The design also allows a single hardware timer to support these features, freeing other timer hardware for application use. Clock measurement is tick-less, removing the periodic clock overflow interrupts. While the implementation is for a GNAT bare-board run-time environment, the presented design principles should be applicable for other systems. Performance tests are done to find the additional overhead to context switches and interrupt handling caused by execution time control. In addition to execution time measurement for interrupts we also provide an interrupt timer, and extend the object-oriented real-time framework to facilitate execution-time control for interrupts. An example application using this feature is given.

Keywords: Ada 2012, execution time control, interrupt clocks, real-time, embedded, GNAT.

1 Introduction

Scheduling analysis of real-time systems rely on the worst-case execution time (WCET) of tasks being known. However, finding the WCET of an algorithm may be hard, for some cases it is not even possible to predict if an algorithm will ever halt [1]. Furthermore, pipelines, caches and other performance enhancing techniques used on contemporary computer architectures makes the WCET even harder to find [2]. This makes WCET analysis a costly and time consuming process. Also, the WCET will often be considerably longer than the average execution time as it includes the very unlikely event of many or all of the performance enhancing techniques failing. Therefore pessimistic scheduling is needed in order to provide an offline guarantee that all hard deadlines will be met, which again leads to poor processor utilization if there are not enough tasks with soft, or no, deadlines to use the remaining processor resources.

Execution time control is a simple, yet powerful tool that allows the total time a task has been executed on a processor to be measured, and a handler to be called when this execution time reaches a specified timeout value. Combined with a scheduling policy taking advantage of this feature, it allows online control of task execution time instead of relying solely on offline guarantees [3]. Execution time control also allows execution time servers such as the deferrable and sporadic server for soft sporadic tasks [4]. Furthermore, it facilitates tasks executing algorithms were there is an increasing reward with increased service (IRIS) [5]. In this case the algorithm is stopped when it has converged or its execution time budget is exhausted. If no acceptable result was computed in time a simpler algorithm may be executed.

Execution time control was standardized together with other new real-time features in Ada 2005 [6]. The standard did not state which execution time budget, if any, that is to be charged the execution time of interrupt handlers. All implementations known to the authors up to this point charged the running task this execution time [7, 8, 9, 10]. This causes inaccuracy to execution time measurement and was pointed out as an issue when the new Ada 2005 real-time features were evaluated [11]. The authors at NTNU have ported GNATforLEON [12], a bare-board run-time environment supporting the Ravenscar restricted tasking model, to the Atmel AVR32 UC3 microcontrollers series [13] and developed it further [14]. When Ada 2005 execution time control was implemented for this run-time environment, special execution time clocks for interrupts handling were added, one for each interrupt priority [15, 16]. This improved accuracy of execution time measurement for tasks and also allowed execution time control for interrupts. These features were presented by the authors at IRTAW 14 and suggested added to Ada 2012 [17]. At the same workshop the developers of MaRTE suggested measuring the execution time of all interrupt handling combined [18]. The workshop decided to suggest execution time measurement both for separate interrupt IDs and all interrupts combined to be added to Ada 2012 [19, 20]. These features are now included in the working draft for the Ada 2012 standard [21].

In this paper there is first a brief presentation of the Ada 2012 execution time control. Then follows an abstraction for clocks and alarms supporting both the real-time clock and timing events, and execution time clocks and timers for tasks and interrupts. It is shown how this design is implemented on the AVR32 UC3 microcontroller series, and performance test results are presented. After this, it is described how execution time control for interrupts is integrated into the object-oriented real-time framework, and an example application is given. Finally there is a discussion on the design and implementation, the implementation cost compared to the benefits of execution time control, the portability of the design, and the real-time framework extensions.

Listing 1: Interrupt execution time clocks

```
package Ada.Execution_Time is
   ...

   Interrupt_Clocks_Supported : constant Boolean
      := implementation-defined;

   Separate_Interrupt_Clocks_Supported : constant Boolean
      := implementation-defined;

   function Clock_For_Interrupts return CPU_Time;

   ...
end Ada.Execution_Time;

package Ada.Execution_Time.Interrupts is

    function Clock
      ( Interrupt  : Ada.Interrupts. Interrupt_Id )
         return CPU_Time;

    function Supported
      ( Interrupt  : Ada.Interrupts. Interrupt_Id )
         return Boolean;

end Ada.Execution_Time.Interrupts;
```

2 Ada 2012 real-time features

2.1 Execution time measurement and timers

The package Ada.Execution_Time defines the type CPU_Time representing elapsed execution time measurement and the function Clock to get the execution time of a task [21]. The execution time of a task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf [21]. For Ada 2005 it was implementation defined which task, if any, that was charged the execution time used by interrupt handlers and run-time services on behalf of the system. Ada 2012 has the ability to account for either the total or separate execution time of interrupts handlers. Listing 1 shows the additions to the specification of Ada.Execution_Time and its new child package Interrupts to support this feature.

The constant Interrupt_Clocks_Supported indicates if the system supports measuring the total execution time of interrupt handlers by the use of the function Clock_For_Interrupts. The function will raise Program_Error when called if not supported. The constant Separate_Interrupt_Clocks_Supported indicates if the system supports measuring the execution time of interrupt handlers separately by the child package Interrupts. In this child package the function Clock returns the execution time for the handler of the given interrupt or raises Program_Error if separate execution time for interrupts is not supported. If Supported returns false for the given interrupt Clock is to return a CPU_Time equal to Time_Of (0).

2.1.1 Timers

The child package Ada.Execution_Time.Timers defines the tagged type Timer which is used for detecting execution time overruns for a single task. The type Timer_Handler identifies a protected procedure to be executed when the timer *expires*. Handlers are set to expire at a given execution time or after a given time interval using two overloading Set_Handler procedures, and may be cancelled using the procedure Cancel_Handler. The function Time_Remaining returns the time remaining until the timer expires. Implementations are allowed to limit the number of timers possible for a single task and raise Timer_Resource_Error if this limit is exceeded. In this work there is a limit of one timer for each task as this limitation is recommended for use with the Ravenscar profile [9]. The Ravenscar profile does however not allow timers, so by including these strict compliance with the profile is lost.

2.2 The real-time clock and timing events

The package Ada.Real_Time defines the types Time and Time_Span used for the monotonic real-time clock, and the function Clock to retrieve the value of this clock. The real time clock corresponds to the passing of physical time, either with the time of system initialization as epoch or another reference time frame.

2.2.1 Timing events

The child package Ada.Real_Time.Timing_Events defines the tagged type Timing_Event that allows protected procedures to be called at a specified time without the need for a task or delay statement. The type Timing_Event_Handler identifies a protected procedure to be executed when the timing event *occurs*. With the exception of the function Time_Of_Event returning the absolute time of the event instead of the time remaining, timing events are used in the same way as timers. Implementations are required to document the upper bound on the overhead of the handler being called. The Ravenscar profile only allows timing events declared at library level.

3 Implementation

3.1 Design

The functionality of the real-time clock (RTC) and execution time clocks (ETCs) are quite similar: both clocks support high accuracy measurement of the monotonic passing of time since an epoch, and both support calling a protected handler when a given timeout time is reached. The main difference is that the RTC is always active, while an ETC is active only when its corresponding task or interrupt is executed. The similarities allow a design where one implementation of clocks and alarms in the internal package System.BB.Time provides support for both execution time control and the real-time features. In addition alarms are used internally for real-time task delay.

The package System.BB.Time defines the type Time to represent the passing of time since the epoch as a 64-bit modular integer, and the type Time_Span as a 64-bit integer with range from -2^{63} to $2^{63} - 1$ to represent time differences. The package defines the limited private types Clock_Descriptor and Alarm_Descriptor to represents clocks and alarms respectively, and Clock_Id and Alarm_Id as access types for these. The private definitions of clocks and alarms are shown in Listing 2.

The package also defines public routines for clock and alarm operations used by the Ada 2012 execution time control and

Listing 2: Definition of clocks and alarms

```
type Clock_Descriptor is
   record
      Base_Time : Time;
      -- Base time of clock

      First_Alarm : Alarm_Id;
      -- Points to the first alarm of this clock

      Capacity : Natural;
      -- Remaining alarm capacity, no more alarms if zero

   end record;

type Alarm_Descriptor is
   record
      Timeout : Time;
      -- Timeout of alarm when set

      Clock : Clock_Id;
      -- Clock of this alarm

      Handler : Alarm_Handler;
      -- Handler to be called when the alarm expires

      Data : System.Address;
      -- Argument to be given when calling handler

      Next : Alarm_Id;
      -- Next alarm in queue when set, null otherwise

   end record;
```

real-time packages. These are also used by the internal package System.BB.Threads for thread wake-up. In addition there are procedures for changing the active execution time clock used by System.BB.Interrupts , System.BB.Protection and the context switch routine. The routines are described in more detail in the following.

3.2 Hardware timer

The 32-bit COUNT / COMPARE system registers of the Atmel AVR32 architecture are used as hardware timer in this work. The COUNT register is reset to zero at system start-up and is incremented by one every CPU clock cycle. The COMPARE interrupt is triggered when COUNT equals COMPARE, cleared when COMPARE is written, and disabled when COMPARE is zero, which is also the reset value of the register. For newer UC3 revisions the COUNT register is reset on COMPARE match, which is not desirable for our use. It is however possible to disable this behavior in the CPU configuration register.

Three hardware timer operations are provided in the package System.BB.CPU_Primitives and implemented using in-line assembler code. The function Get_Count returns a snap-shot value of COUNT. The procedure Adjust_Compare sets COMPARE according to the argument C while preventing that the interrupt is lost:

$$\text{COMPARE} \leftarrow max(C, \text{COUNT} + \epsilon)$$

Here ϵ is a small number of clock cycles, so that an interrupt will be pending immediately after leaving the procedure if C was less than COUNT. The procedure Reset_Count sets COUNT to zero and returns the previous COUNT value c_p in one atomic operation:

$$c_p \leftarrow \text{COUNT}_- \quad (1)$$
$$\text{COUNT}_+ \leftarrow 0 \quad (2)$$

This is done by two instructions, the first reading c_p from COUNT, the second writing the value 2 to COUNT as this is the number of clock cycles the two instructions take. The operation is done atomically as interrupts are disabled when executing kernel calls. No clock cycles are lost when resetting the COUNT register: the sum of c_p and COUNT equals the value COUNT would have had without reset. The COMPARE register is not altered by the reset procedure, and has to be updated with a call to Adjust_Compare if needed.

3.3 Clocks

The type Clock_Descriptor seen in Listing 2 represents clocks and has three data members: (1) The Base_Time that holds the part of the clocks elapsed time not present in the hardware timer. It is initialized to zero. (2) The First_Alarm pointing to the first set alarm of the clock. It is initialized to a sentinel alarm and is never **null** after this. (3) The Capacity gives the remaining number of alarms allowed for this clock. For the real-time clock it is initialized to Natural'Last which in practice means no limit on the number of alarms. For task clocks Capacity is initialized to one as is recommended for the Ravenscar profile [9]. We also allow one alarm for interrupt clocks for interrupts not of the highest interrupt priority.

The package body has Clock_Descriptors for the RTC, interrupt clocks, and the internal idle clock used when the system is executing the idle-loop. In order save memory there is a pool of interrupt clocks and a look-up table with Interrupt_ID as index, instead of having a Clock_Descriptor for every interrupt. This Ravenscar run-time environment is designed not to use dynamic memory in the kernel [12]. The pool size is set to allow at most ten interrupts, but this can be easily be changed in the package System.BB.Parameters. The Clock_Descriptor of threads is stored in the type Thread_Descriptor of the package System.BB.Threads.

3.3.1 Clock management

After initialization of the package there are precisely two active clocks: the RTC that is always active and the ETC that points either to the clock of the running thread, to the clock for the interrupt being handled or to the idle clock. The ETC is changed as a result of a context switch, interrupt handling, or system idling.

The low-level interrupt handler of the run-time environment calls Enter_Interrupt with the Interrupt_ID prior to calling the interrupt handler. This procedure pushes the current ETC on a stack and activates the interrupt clock found in the look-up table as the new ETC. After the interrupt handler is called a call to Leave_Interrupt pops and reactivated the old ETC. The interrupt handler may also be interrupted by a higher priority

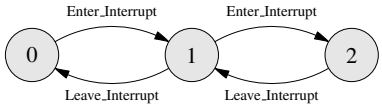

Figure 1: Stack states with two interrupt levels.

interrupt as seen in Figure 1. The stack size is limited by the systems number of interrupt levels.

There is no idle thread in the run-time environment. Instead the thread τ_a that finds the ready queue empty when leaving the kernel enters an idle-loop waiting for any thread to be made runnable by an interrupt. Prior to entering the idle loop a call to Enter_Idle activates the idle clock as the ETC. If τ_a is made runnable it calls Leave_Idle to reactivate its clock. Also a context switch may take place and change clock to the new running thread τ_b as seen in Figure 2. When τ_a resumes execution the idle clock will be activated by the context switch again instead of the task clock. In order to do this the Thread_Descriptor has a field Active_Clock that points either to the tasks own clock, or the idle clock if the tasks is executing the idle loop. Only one thread at a time will enter the idle loop.

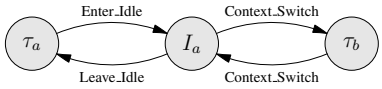

Figure 2: System idling with two tasks.

The states in Figure 2 are sub-states of state 0 in Figure 1, any of the states can be interrupted and will be restored when the interrupt handler is left. Since no task can have a base priority in the interrupt priority range in the Ravenscar profile context switches can only occur in state 0, after the task priority has be lowered back to the tasks base priority.

3.3.2 Measuring time

The use of the hardware timer is tick-less and therefore does not require a periodic clock overflow interrupt. Instead COUNT is reset using Reset_Count when the ETC is changed, and the base time of the RTC and the old ETC is incremented with the previous COUNT value c_p. By doing this the same hardware timer may be used for both the RTC and the ETC as seen in Figure 3.

The elapsed time of a clock t since the epoch is retrieved by the function Elapsed_Time, and is computed from the base time b and the COUNT register value:

$$t = \begin{cases} b + \text{COUNT} & \text{if clock is active} \\ b & \text{else} \end{cases}$$

An interrupt may occur after reading the base time but before reading COUNT in Elapsed_Time. This will update the base time and reset COUNT, making the sum of the earlier read base time and COUNT invalid. To avoid this there is a check

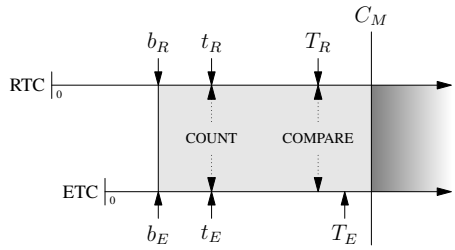

Figure 3: Relation between the RTC and ETC, and the hardware timer registers. The base time of the clocks are aligned.

after reading COUNT to see if the base time has been updated, in which case the updated base time will be returned as the elapsed time.

3.3.3 Setting the hardware timer

The COMPARE register is adjusted after updating ETC or after changing the first alarm of an active clock. If within the light gray region in Figure 3 the value C given to Adjust_Compare is the smallest difference d for the RTC and ETC between the first timeout T of the clock and its base time b:

$$C = min(min(d_R, d_E), C_M)$$

In rare cases b may be slightly larger than T. To handle this so that the COMPARE interrupt will be pending immediately after calling Adjust_Compare and prevent overflow d is computed as:

$$d = T - min(T, b)$$

Correct time measurement depends on COUNT never overflowing and C_M is a safety mechanism to prevent this critical error. By having $C_M = (2^{32} - 1) - C_S$ there will always be a pending COMPARE interrupt the last C_S clock cycles before overflow. This region is marked darker gray in Figure 3. If interrupts are not blocked by the system for longer than C_s the interrupt will be handled and COUNT reset when Enter_Interrupt is called, preventing overflow. The COMPARE interrupt handler will simply ignore this "false" interrupt. We use a large safety region $C_s = 2^{31}$ to provide ample time for the interrupt to be handled.

3.4 Alarms

The type Alarm_Descriptor seen in Listing 2 is used for representing internal alarms and has five data members: (1) Timeout that gives the time of event when set. (2) The Clock of the alarm given as argument to the alarm initialization procedure. If the Capacity for the clock is zero the initialization will not succeed and the alarm cannot be used. (3) Handler which is an access to the procedure that is called when the alarm expires and (4) the argument Data of type System.Address given when calling this handler. The handler and data are set during initialization of the alarm and remain constant after this. (5) The access Next pointing to the next alarm in the queue when the alarm is set, **null** otherwise.

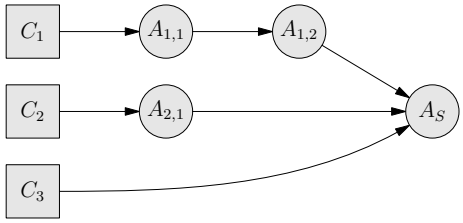

Figure 4: Three clocks set with two, one and zero alarms in addition to the sentinel at the end of the queue.

3.4.1 The alarm queue

The queue of pending alarms for clocks is managed as a linked list sorted in ascending order after the Timeout value of the alarms. Alarms with equal Timeout value are queued in FIFO order. To avoid the special condition of an empty queue there is a sentinel alarm with timeout at Time'Last that is always present at the end of the queue. The constant Time_Last seen by the user is set to Time'Last − 1 so that the sentinel is always last. This avoids an additional check when searching the queue. One sentinel alarm without handler is shared between all clocks as shown in Figure 4 to save memory.

The procedure Set takes the alarm and timeout as argument, sets the timeout field of the alarm, and searches the queue of the clock associated with the alarm for another alarm with timeout greater than the Timeout, the alarm is then inserted before this one and always before the sentinel. The procedure Cancel first checks that the alarm is set, and if so searches for the alarm in the queue and removes it. It is necessary to search the queue since to find the alarm before the one being removed as it is implemented as a single linked list. Both procedures reprogram the hardware timer if the alarm inserted or removed is first in the queue of an active clock.

3.4.2 Calling alarm handlers

The COMPARE interrupt handler has the highest interrupt priority. When this handler is called the procedure Alarm_Wrapper is called first for the RTC and then for the interrupted ETC on top of the stack. At this point the active ETC is that of the COMPARE interrupt itself, for which no alarms are allowed, so only the interrupt ETC on top of the stack or the RTC may be the cause of the interrupt. As the wrapper is called for both clocks there is no need to check which caused the interrupt. The alarm wrapper removes all alarms with timeout less or equal to the base time of the clock from the head of alarm queue one at the time, clears the alarm and calls the handler with the data as argument. The alarm handler can, and very often will, alter the alarm queue, so it is important to have the queue in a consistent state before calling the handler and reread the first alarm of the clock after calling the handler.

3.5 Ada 2012 interface

The implementation of the application programming interface as described by the Ada reference manual [6] is quite similar for the real-time and execution time control features as they use the same internal time, clock and alarm types.

Listing 3: Interrupt timer specification

```
package Ada.Execution_Time.Interrupts.Timers is

   type Interrupt_Timer (I : Ada.Interrupts.Interrupt_ID)
      is new Ada.Execution_Time.Timers.Timer
     (Ada.Task_Identification.Null_Task_Id'Access)
     with private;

private

   type Interrupt_Timer (I : Ada.Interrupts.Interrupt_ID)
      is new Ada.Execution_Time.Timers.Timer
     (Ada.Task_Identification.Null_Task_Id'Access)
     with null record;

end Ada.Execution_Time.Interrupts.Timers;
```

3.5.1 Clocks

The functions named Clock in the packages Ada.Real_Time, Ada.Execution_Time and Ada.Execution_Time.Interrupts all call Elapsed_Time with the Clock_Id of the RTC, a task clock or an interrupt clock as argument respectively. If there is no internal clock for a given interrupt CPU_Time_First is returned. To get the total execution time spent on interrupt handlers interrupts Clock_For_Interrupts iterates through all Interrupt_IDs and finds sum of calling Clock for each.

3.5.2 Timing events and timers

The tagged types Timing_Events and Timer both have an Alarm_Descriptor, an Alarm_Id that points to this after initialization and a user handler of type Event_Handler and Timer_Handler respectively. Both types use their alarm to call a wrapper with the object as argument, that again calls the user handler. The difference is in the initialization of the alarm where Timing_Events use the RTC, while Timer uses the execution time clock of the task. The alarm initialization may fail for Timer in which case the exception Timer_Resource_Error will be raised. For Timing_Event the initialization is asserted to succeed.

For both types the procedure Set_Handler first calls Cancel of System.BB.Time to remove the alarm from the queue if necessary before it sets the user handler and calls Set if this handler is not **null**. This has to be done as Set expects the alarm to be cleared. The procedure Cancel_Handler checks if the user handler is set in which case Cancel is called and Cancelled is set to true. Operations are done atomically by using the package System.BB.Protection for blocking interrupts.

3.5.3 Interrupt timers

To allow execution time control for interrupts the non-standard package Ada.Execution_Time.Interrupts.Timers shown in Listing 3 defines the tagged type Interrupt_Timer that inherits Timer and its operations. Note that the constant Null_Task_Id from Ada.Task_Identification has to be marked **aliased** to be used as discriminant when inheriting Timer. No body is needed for this package. The initialization procedure for timers checks if the object is of type Interrupt_Timer in which case it uses the interrupt clock instead of task clock. Interrupt timers are used in the exact same way as task timers.

Table 1: Performance test results in CPU cycles

Test	Implementation		
	TI-ETC	T-ETC	N-ETC
Context switch	602	602	471
Timing event	381	272	270
Interruption cost	296	503	–

4 Performance

Performance testing of the implementation is done with the Atmel AVR32 UC3A0512 microcontroller on the EVK1100 evaluation board. For the tests the microcontroller is run at 60 MHz, and is programmed and debugged using the Atmel JTAG ICE Mk II. Test data is sent over the serial line to the PC where it is retrieved and analyzed using GNU Octave.

The implementation with support for task and interrupt execution time control (TI-ETC) is tested against two other versions of the run-time environment: one where support for execution time control is completely removed (N-ETC), and one that supports execution time control for tasks only (T-ETC). Here N-ETC use the COUNT / COMPARE registers for the RTC in the same way as TI-ETC, with the exception of COUNT being reset in the COMPARE handler. This means that it has zero additional overhead to context switches and interrupt handling. For T-ETC the difference from TI-ETC is that the interrupt clocks and corresponding packages are removed, together with the calls to Enter_Interrupt and Leave_Interrupt in the low-level interrupt handler. This implementations should have zero additional overhead to interrupt handling compared to N-ETC, but the same additional overhead as TI-ETC for context switches.

4.1 Context switch overhead

The purpose of this test is to find the overhead to context switches by changing the execution time clock. We test without an alarm being set for the clock as the overhead is found to be the same regardless of alarm status. The test is done by having a task τ_a release a higher priority task τ_b that is blocked on an entry of a protected object. The release time is read by the protected procedure opening the entry, and is returned to τ_b by the entry. After being released τ_b reads the clock and the two time values are transferred over the USART line before the task blocks again and the test is repeated.

The first row in Table 1 shows the results for the implementations. The exact same number of clock cycles was measured in all samples for this test. This is due to simplicity of the executed test program and the deterministic nature of the UC3 microcontroller. The additional overhead caused by execution time control is inferred to be 131 clock cycles or 2.2 μs at the clock frequency used in the test.

4.2 Timing event overhead

The system is required to document the overhead of handling timing event occurrences. This is also a good measure of interrupt handling overhead in general caused by execution time control. The program has a single timing event that is programmed to occur with random intervals between 1 and 3 milliseconds. When the handler is called the difference between the timeout and the clock is recorded. After 100 samples the data is transferred over the USART line and the test is repeated.

The second row in Table 1 shows the results for the implementations in clock cycles. As before there was only one measured overhead value for each implementation due to the simplicity of the test program and the determinism of the UC3. It is inferred from the results that execution time control gives an additional overhead of 111 clock cycles to interrupt handling, or 1.85 μs at the clock frequency used for the test. The difference of two clock cycles between T-ETC and N-ETC is inferred to be caused by small differences in the function Elapsed_Time reading the real-time clock.

4.3 Cost to interrupted task

The execution time cost to the task being interrupted is greater than zero, as the interrupt clock is activated by the low-level interrupt handler, and not by hardware. The purpose of this test is to find this cost. The test is done by having a single task τ first setting a timer for its own execution time clock to expire in 20 ms if this timer is not already set, then reading its execution time clock, busy waiting 10 millisecond, and then reading this clock again. The clock values are transferred over the USART line and the test is repeated. Only the interrupt caused by the timer can occur between the two clock readings, and it can occur only once. A protected procedure with **null** as the only statement is used as handler. To find the cost we compare the difference in execution time when interrupted to when the task is not interrupted. This test is only relevant for TI-ETC and T-ETC.

The last row in Table 1 shows the cost to the interrupted task in clock cycles for the implementations with and without separate execution time clocks for interrupts. The execution time when not interrupted was always the same number of clock cycles for both implementations due to the deterministic nature of the UC3 microcontroller. When interrupted the execution time varied with one clock cycle. The worst-case cost of interruption is shown.

5 Use of interrupt timers

To ease development of real-time applications an object-oriented framework has been developed by several contributors in the Ada community [22]. The framework provides common real-time patterns such as periodic and sporadic tasks, detection of deadline miss and overrun detection, execution-time servers and more. By integrating the non-standard Interrupt_Timer into this framework it is also possible to control the execution-time spent on interrupt handling and thereby prevent deadlines being lost due to bursts of interrupts. The framework components related to interrupt handling can be separated into three parts: (1) the interface Interrupt_Controller used to control hardware interrupt generation; (2) the protected interface Interrupt_Server used to control the execution time spent handling a given Interrupt_ID

Listing 4: Definition of interrupt controller

```
package Interrupt_Controllers is

   type Interrupt_Controller is limited interface;

   procedure Enable
     (C : in out Interrupt_Controller;
      I : Interrupt_ID) is abstract;

   procedure Disable
     (C : in out Interrupt_Controller;
      I : Interrupt_ID) is abstract;

   function Supported
     (C : Interrupt_Controller;
      I : Interrupt_ID) return Boolean is abstract;

   type Any_Interrupt_Controller is
     access all Interrupt_Controller'Class;

   Unsupported_Interrupt : exception;

end Interrupt_Controllers;
```

Listing 5: Interrupt server interface

```
package Interrupt_Servers is

   type Interrupt_Server_Parameters is tagged
      record
         Controller : Any_Interrupt_Controller;
         Budget : Time_Span;
      end record;

   type Interrupt_Server is protected interface;

   procedure Initialize
     (S : in out Interrupt_Server) is abstract;

   type Any_Interrupt_Server is access all Interrupt_Server;

end Interrupt_Servers;
```

Listing 6: Deferrable interrupt server specification

```
package Interrupt_Servers.Deferrable is

   type Deferrable_Server_Parameters
     is new Interrupt_Server_Parameters with
      record
         Period : Time_Span;
      end record;

   protected type Deferrable_Interrupt_Server
     (I     : Interrupt_ID;
      Param : access Deferrable_Server_Parameters) is
      new Interrupt_Server with

      procedure Initialize;

      pragma Priority (Any_Priority'Last);

   private

      procedure Replenish (Event : in out Timing_Event);
      procedure Overrun (TM : in out Timer);

      Replenish_Event : Timing_Event;
      Execution_Timer : Interrupt_Timer (I);
      Next : Time;
      Disabled : Boolean := True;

   end Deferrable_Interrupt_Server;

end Interrupt_Servers.Deferrable;
```

in accordance with some policy; (3) the protected interrupt handlers, the framework provides the release mechanism Sporadic_Interrupt to release tasks as a result of an interrupt.

5.1 Interrupt controller

The interface Interrupt_Controller is defined as shown in Listing 4. The interface will typically be implemented by a peripheral driver. Depending on the peripheral it may control one or more interrupts. Use of the interface is very straight-forward: Enable enables the generation of given Interrupt_ID and Disable disables it. The function Supported indicates if the controller supports the interrupt, if other operations of a controller is called with an unsupported interrupt the Unsupported_Interrupt exception will be raised.

5.2 Interrupt servers

The interface Interrupt_Server shown in Listing 5 uses Interrupt_Controller to control the execution time spent handling a given interrupt according to a policy by enabling and disabling its generation. The tagged type Interrupt_Server_Parameters is used to pass the controller and the execution time budget to implementations of the interface.

The protected object Deferrable_Interrupt_Server shown in Listing 6 and 7 implements this interface following the deferrable server policy. This allows us to model the execution time spent handling the given interrupt as a periodic task with a given period and budget. The type Deferrable_Server_Parameters defines the additional parameters needed by the server, in this case the replenishing period of the execution time budget. Notice that the Interrupt_ID is given as a separate discriminant, this is needed to declare the timer statically in the protected object. Internally the deferrable server has a timing event used to call the procedure Replenish periodically with the period given as parameter. The procedure sets the execution time budget for the interrupt using the interrupt timer, and enables the interrupt if necessary. The first call to Replenish is at the system epoch, and will enable the generation of the interrupt. The procedure Overrun is called when the execution time budget is exceeded and disables the generation of the interrupt.

5.3 Example application

Our example application has a real-time task implemented by a tagged type inheriting Periodic_Task of the real-time framework. The task has period 10 ms and a 5 ms budget, and we use the periodic release mechanism with overrun and deadline miss detection. For each release the task simply busy waits 75% of its budget.

In addition the application receives data from the PC through the USART line. We use the same hardware setup as for the performance tests. The tagged type USART_Controller implements Interrupt_Controller and is used to setup, enable and

Listing 7: Deferrable interrupt server body

```
package body Interrupt_Servers.Deferrable is

   protected body Deferrable_Interrupt_Server is

      procedure Initialize  is
      begin
         pragma Assert (Param.Controller.Supported (I));
         Next := Epoch;
         Replenish_Event.Set_Handler
           (Next, Replenish'Access);
      end  Initialize ;

      procedure Replenish (Event : in out Timing_Event) is
      begin
         Execution_Timer.Set_Handler
           (Param.Budget, Overrun'Access);
         if  Disabled then
            Disabled := False;
            Param.Controller.Enable (I);
         end if;
         Next := Next + Param.Period;
         Event.Set_Handler (Next, Replenish'Access);
      end Replenish;

      procedure Overrun (TM : in out Timer) is
      begin
         pragma Assert (not Disabled);
         Disabled := True;
         Param.Controller.Disable (I );
      end Overrun;

   end Deferrable_Interrupt_Server;

end Interrupt_Servers.Deferrable;
```

Listing 8: Usage of interrupt server

```
package body Test is

   USART : aliased USART_Controller (USART_1_Address);

   Param : aliased constant Deferrable_Server_Parameters
     := (Controller  => USART'Access,
         Budget      => Milliseconds (1),
         Period      => Milliseconds (10));

   USART_Server : Deferrable_Interrupt_Server
     (USART_1, Param'Access);

   protected RX_Counter is
      pragma Interrupt_Priority (USART_1_Priority);
      function Get_Count return Natural;
   private
      procedure Increment;
      pragma Attach_Handler (Increment, USART_1);
      Count : Natural := 0;
   end RX_Counter;

   protected body RX_Counter is
      function Get_Count return Natural is
      begin
         return Count;
      end Get_Count;
      procedure Increment is
      begin
         USART.Clear (USART_1);
         Count := Count + 1;
      end Increment;
   end RX_Counter;

   procedure Run is
      Next :  Time := Epoch;
   begin
      loop
         delay until  Next;
         Put (RX_Counter.Get_Count);
         New_Line;
         Next := Next + Seconds (1);
      end loop;
   end Run;

begin
   USART.Initialize;
   USART_Server.Initialize;
end Test;
```

disable the RX interrupt of the USART. A protected object with the USART interrupt handler counts the number of characters received. The environment task outputs this count every second. This task has lower priority than the real-time task and no deadline.

The baud rate of the USART line is a far higher rate than the system is able to receive using interrupts. However, the *intended* usage is that characters are typed one-by-one to the serial line by the user, and therefore will be limited to a few characters per second. Since we do not fully trust this limitation to be respected, a deferrable interrupt server is included to control the execution time spent handling receive USART interrupt. We let the server have a replenishing period of 10 ms, the same period as the real-time task, and a budget of 1 ms. Hence, the total utilization not considering the background task, is 60% which is known to be safe using RMA. The parts of the application related to interrupt handling are shown in Listing 8.

Running on the UC3A0512 of the EVK1100 evaluation board, the application correctly counts each character sent by typing in the serial communication program "minicom". In order to test the interrupt execution time control, we use the "cat" command to write the entire source code of the application to the serial device file, and observe that the USART interrupt is disabled when the budget is exceeded and re-enabled when it is replenished. During the test the real-time task did not miss any deadline. However, only 40% of the characters sent were successfully received by the system. This loss could be prevented by using USART hardware flow control or buffering, but we want to keep the example application simple. As expected the real-time task misses all its deadlines during the burst when the interrupt server is removed from the system.

6 Discussion

6.1 Design and implementation

Our design supports both the real-time clock and timing events, and execution time clocks and timers using one internal clock and alarm implementation. This removes most of the near duplicate code compared to separate implementations. Table 2 shows code metrics for the implementations with full, task only and no execution time control as reported by the "gnatmetric" tool. Only packages that are different for

Table 2: Code metrics for implementations

Implementation	Decl.	Stat.	SLOC
TI-ETC	243	516	759
T-ETC	221	473	694
N-ETC	158	264	422

the implementations are included. As seen the difference between full and task only execution time control is small, only 65 logical code lines which includes two additional packages for interrupt clocks and timers. For System.BB.Time the difference is only 11 logical code lines. The difference between full and no execution time control is greater, 337 logical code lines, but this includes seven additional packages for execution time control. For System.BB.Time the difference is only 27 logical code lines. Overall the number of code lines added by execution time control seems small and acceptable compared to the features provided.

Another benefit of our design is that one hardware timer is sufficient to support both the RTC and the ETC. By using only one hardware timer and one clock interrupt, our system is easier to understand and debug as there are no race conditions between interrupts of different hardware timers that need to be handled. The reduced hardware requirements for the run-time environment also frees timers for the application. Compared to the earlier implementation of execution time control [16] that used one of the two Timer / Counter hardware timer units of the UC3A microcontroller, both these are available for the application with the new design and can be used for pulse-wave modulation (PWM), external signal generation and more.

The tick-less design means that there are no periodic clock interrupts to increment the most significant part (MSP) part of the time value. If context switches and interrupts occur more often than C_M which is 35.8 seconds on our system running at 60 MHz, there will be no interrupts caused by clock measurement. For typical real-time systems there will be more frequent context switches and interrupts than this. The execution time of the clock overflow handlers may not be negligible, meaning that it could affect scheduling analysis. While the tick-less design comes at the cost of additional overhead to context switches and interrupt handling, the benefits of removing the periodic clock tick is greater.

6.2 Portability

While our design is implemented on the AVR32 UC3 microcontroller series, it should be portable to any architecture where it is possible to implement the routines Get_Count, Adjust_Compare and Reset_Count according to their specification. With minor modifications it should also be possible to use 16-bit hardware timers instead of the 32-bit timer used in this paper. In this case it would be necessary to reduce the clock resolution as overflow interrupts would occur every 546 μs at the resolution of 60 MHz used in this paper.

Our implementation uses a hardware timer within the processor core, giving the benefit of a deterministic, constant access time. It is possible to use a peripheral hardware timer, although it may be harder to implement Reset_Count without clock cycle leakage as the access time for reading and writing timer registers over the peripheral bus would not be constant for most systems.

6.3 Overhead caused by switching clocks

The two overhead tests measure the time it takes either between two clock readings, or the time between an event taking place at a known time and reading the clock. It is known whether this time includes changing execution time clocks or not for the implementation being tested. When comparing results it is important to remember that there are minor changes in the compiler output that affect the result, and that the function reading the clock also has minor changes between the implementations with and without execution time control. However, the main difference in overhead is caused by changing clocks and the results are considered valid. The context switch and interrupt handling overhead was found to be 131 and 111 clock cycles respectively. The small difference of 20 clock cycles between the two results is due to differences in clock management.

The additional overhead to context switches and interrupt handling caused by the full implementation is significant. At the clock frequency of 60 MHz used in the tests this additional overhead is 2.2 μs and 1.85 μs respectively. This adds to the latency for interrupt handlers and task release, and reduces the overall system performance. Still, the overhead is not prohibitively high taking into account the benefits provided by execution time control. Also, this overhead includes the cost of the tick-less timer that removes the overhead to tasks and interrupts caused by the periodic clock interrupt.

6.4 Cost of interruption

The test measuring the execution time cost to a task being interrupted is more accurate than the overhead tests as we compare the difference when the interrupt did and did not happen for the same implementation. By design we know that at most one interrupt may occur between reading the clocks. The cost of interruption to the task when using interrupt clocks was 297 or 298 clock cycles. When using the clock of the interrupted task the cost was 502 or 503 clock cycles. The difference between the two implementations is thus 205 clock cycles, but without interrupt clocks the cost includes the whole execution time overhead of calling the timer handler including the Alarm_Wrapper and Execute_Handler procedures. The cost would be less if an ordinary interrupt handler was used.

The small but noticeable cost to the interrupted task when using interrupt clocks means that if a task is interrupted many times its budget may have to be extended to allow for this. Without interrupt clocks the cost of interruption is varying, depending on what is done in the interrupt handler. In the case of very simple handlers this cost may even be lower than when using interrupt clock due to the overhead of changing clocks. Still, having a constant cost regardless of what is done in the handler is better for analysis. It is also possible to transfer execution time from the task clock to the interrupt

clock before and after calling the interrupt handler to refund the tasks cost without its clock going backwards observably. While this scheme would reduce the cost to interrupted tasks, it would increase the complexity and also would need to be tuned depending on compiler output, and was therefore discarded.

6.5 Hardware support

Ideally we would like to have near zero overhead to context switches and interrupt handling caused by execution time control, and near zero cost of interruption for tasks. This is not feasible without a specialized hardware timer that allows execution time clocks to be changed more efficiently. Therefore the authors have designed a Time Management Unit (TMU) supporting 64-bit timer values and atomic clock changes [23]. It is designed to have a simple memory mapped interface accessible though the peripheral bus, making it portable to different architectures. The TMU has been implemented with the AVR32 UC3 core as a part of a masters thesis at NTNU in cooperation with Atmel Norway [24]. Simulation results indicates that the overhead of switching clocks can be reduced to less than 50 clock cycles by using this hardware timer.

6.6 Interrupt timer

The interrupt timer is not a part of the Ada 2012 standard but should in the authors opinion be added to the next revision for the following reasons. First it provides execution time control for interrupts similar to that for tasks. If we measure the execution time for interrupts it should also be controllable by means such as the framework extensions described in this paper. This is important as the execution time spent handling interrupts may be very hard to predict as the interrupts may be generated by external hardware that are not controlled by the application. Alternatives to interrupt timers are to count the number of interrupts and disable the interrupt if the count gets to high, or to poll the execution time of the interrupt after the handler is called and disable the interrupt if the budget is exceeded. These solutions are less precise and also less efficient than using interrupt timers.

Also, the cost of including interrupt timers is small for our implementation as the same clock abstraction and hardware timer is used for task and interrupts. Since the interrupt timer inherits the operations from the task timer, no additional code is needed other than the definition of the tagged type and the code to initialize interrupt timers.

6.7 Framework extensions

The interrupt timer allows us to extend the object-oriented real-time framework to also provide execution time servers for interrupts following the same pattern as used for task execution time servers. While the task server controls the execution time for a group of tasks released sporadically, the interrupt server controls the execution time spent invoking one interrupt handler many times. The object-oriented nature of the framework allows us to create servers suitable for different needs. We have implemented the deferrable server under the assumption that it is acceptable to ignore interrupts for a while, but other schemes may for instance be to reconfigure the system into fail-safe mode in the case of interrupt overruns.

The deferrable interrupt server has a budget that is replenished periodically, and disables interrupt generation if this budget is exceeded. Since there is no way to cancel the interrupt being handled in Ada, the budget has to allow for an overrun of one additional handler invocation for the cases where the budget is exceeded right after entering the low-level handler. It should be considered adding a user handler that is called to notify the application when an interrupt is disabled, to allow for instance hardware diagnostics. This could of course also be done in the Disable procedure of the peripheral driver. In this case it could be useful to add a cause argument to this procedure.

6.8 Example application

The example application is typical in that we must assume one rate of interrupts, but cannot guarantee it as the generation the interrupt is not controlled by the application. Burst of interrupts may also be caused by permanent or transient hardware faults. The result is that the system has to handle more interrupts than budgeted for in the real-time analysis, if the effects of interrupt handling was analyzed at all. This could cause deadlines to be missed and thereby system failure. The presented extensions to the real-time framework provides an easy way to protect our real-time application against these situations.

In the example application we use the USART RX interrupt to receive data sent on the serial line. This is reasonable and efficient given that we know that the characters are sent by the user typing in a serial communication program. However the high baud rate means that the system could be overloaded with interrupts if this limitation is not respected. By using the deferrable interrupt server of the real-time framework we can easily set a budget for the interrupt so that our real-time task is guaranteed sufficient execution time to meet its deadline. No deadlines were lost due to burst of interrupts when the application was tested with the deferrable server, while several deadlines were lost during the burst when the server was not used. This gives a good indication that the deferrable interrupt server works as intended.

7 Conclusion

Our implementation of Ada 2012 execution time control has a design with several benefits. By using a single clock and alarm abstraction to support both the real-time and execution time clocks, we have reduced the amount of code needed for the implementation. This also allows just one hardware timer to support both these clocks, reducing the complexity of the system and the hardware requirements of the run-time environment. This frees valuable hardware timers for the application. We use the hardware timer in a tick-less manner, meaning that there are no periodical clock interrupts. By requiring only one hardware timer the design should also be easy to port to other architectures with similar timers.

Performance testing shows a noticeable overhead to context switch and interrupt handling caused by our implementation

of execution time control. However, this is in our opinion justified by the value of the provided features, and the tickless clock measurement. We also found that there is a low constant execution time cost to tasks being interrupted. While zero cost is the ideal, this constant cost is an improvement in analyzability compared to the varying, and in most cases higher, cost without separate execution time measurement for interrupts.

We have presented an interrupt timer providing execution time control for interrupts similar to that for tasks. This feature is not a part of the Ada 2012 standard where the execution time for interrupts can only be measured, and not controlled. By extending the object-oriented real-time framework using the interrupt timer we provide a deferrable execution time server for interrupts so that the time spent on interrupt handling may be analyzed as a periodic task. The example application shows that our framework extensions provide an easy and elegant solution to prevent deadlines being missed due to bursts of interrupts. In the authors opinion interrupt timers should be added to the next revision of the Ada programming language.

8 Further work

Work is in progress with an implementation using a specialized Time Management Unit (TMU) for execution time control instead of the COUNT / COMPARE timer, and test this implementation with the AVR32 UC3 core in cooperation with Atmel Norway.

References

[1] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, no. 2, 1937.

[2] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[3] A. Wellings and A. Burns, *Ada-Europe 2007*, ch. Real-Time Utilities for Ada 2005, pp. 1–14. Springer Berlin / Heidelberg, 2007.

[4] A. Burns and A. Wellings, "Programming execution-time servers in Ada 2005," in *Proc. 27th IEEE International Real-Time Systems Symposium RTSS '06*, pp. 47–56, Dec. 2006.

[5] C. M. Krishna and K. G. Shin, *Real-Time Systems*. McGraw-Hill International Edition, 1997.

[6] ISO/IEC, *Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1*.

[7] M. G. Harbour *et al.*, "Implementing and using execution time clocks in Ada hard real-time applications," in *Lecture Notes in Computer Science*, vol. Volume 1411/1998, pp. 90–101, Springer Berlin / Heidelberg, 1998.

[8] M. G. Harbour and M. A. Rivas, "Managing multiple execution-time timers from a single task," *Ada Lett.*, vol. XXIII, no. 4, pp. 28–31, 2003.

[9] J. A. de la Puente and J. Zamorano, "Execution-time clocks and Ravenscar kernels," *Ada Lett.*, vol. XXIII, no. 4, pp. 82–86, 2003.

[10] S. Urueña, J. Pulido, J. Redondo, and J. Zamorano, "Implementing the new Ada 2005 real-time features on a bare board kernel," *Ada Lett.*, vol. XXVII, no. 2, pp. 61–66, 2007.

[11] A. Wellings, "Implementation experience with Ada 2005," *Ada Lett.*, vol. XXVII, no 2, pp. 59–60, 2007. session report.

[12] J. F. Ruiz, "GNAT pro for on-board misson-critical space applications," *Ada-Europe*, 2005.

[13] Atmel Corporation, *AVR32UC3 - Technical Reference Manual*, March 2010.

[14] K. N. Gregertsen and A. Skavhaug, "An efficient and deterministic multi-tasking run-time environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, pp. 1572–1575, April 2009.

[15] K. N. Gregertsen, "Execution time management for AVR32 Ravenscar," Master's thesis, Norwegian University of Science and Technology (NTNU), 2008.

[16] K. N. Gregertsen and A. Skavhaug, "Implementing the new ada 2005 timing event and execution time control features on the avr32 architecture," *Journal of Systems Architecture*, vol. 56, pp. 509–522, 2010.

[17] K. N. Gregertsen and A. Skavhaug, "Execution-time control for interrupt handling," *Ada Lett.*, vol. 30, 2010.

[18] M. A. Rivas and M. G. Harbour, "Execution time monitoring and interrupt handlers: position statement," *Ada Lett.*, vol. 30, 2010.

[19] T. Vardanega, M. G. Harbour, and L. M. Pinho, "Session summary: language and distribution issues," *Ada Lett.*, vol. 30, 2010.

[20] S. Michell and J. Real, "Conclusions of the 14th international real-time ada workshop," *Ada Lett.*, vol. 30, 2010.

[21] ISO/IEC, *Ada Reference Manual - ISO/IEC 8652:201x(E) (Draft 13).*

[22] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge, 2007.

[23] K. N. Gregertsen and A. Skavhaug, "Functional specification for a Time Management Unit." Presented at SAFECOMP 2010.

[24] S. J. Søvik, "Hardware implementation of a Time Management Unit," Master's thesis, NTNU, 2010.

A.7 Article No. 7

K. N. Gregertsen and A. Skavhaug:

“Improving the performance of execution time control by using a hardware Time Management Unit”

Accepted for Ada-Europe 2012 [25].

Improving the performance of execution time control by using a hardware Time Management Unit

Kristoffer Nyborg Gregertsen[1] and Amund Skavhaug[1]

Department of Engineering Cybernetics, NTNU
N-7491 Trondheim, Norway
{gregerts,amund}@itk.ntnu.no

Abstract. This paper describes how a dedicated Time Management Unit (TMU) is used to reduce the overhead of execution time control. While the implementation described here is for Ada 2012 and a GNAT bare-board run-time environment, the principles should be applicable to other languages and run-time systems. The TMU has been implemented as a peripheral unit for the Atmel AVR®32 UC3 series of microcontrollers, and test results from simulation with the syntheziable RTL code of this system-on-chip are presented.

1 Introduction

Scheduling analysis of real-time systems relies on the worst-case execution time (WCET) of tasks being known. However, finding the WCET of an algorithm may be very hard, and performance enhancing techniques such as pipelines and caches makes it even harder [22]. This makes WCET analysis a costly and time consuming process. Also, the WCET will often be considerably longer than the average execution time, as it includes the unlikely event of many or all of the performance enhancing techniques failing. Therefore scheduling will often be pessimistic to provide an offline guarantee that all deadlines are met, which again leads to poor processor utilization.

Execution time control allows the total time a task has been executed on a processor to be measured, and a handler to be called when this execution time reaches a specified timeout value. Combined with a scheduling policy taking advantage of this feature, it allows online control of task execution time instead of relying exclusively on offline guarantees [21]. Execution time control also allows execution time servers for soft sporadic tasks [3], and algorithms where there is an increasing reward with increased service (IRIS) [13].

Many systems support execution time control, examples are real-time POSIX [19], real-time Java [16], and Ada since the 2005 revision of the language standard [11]. Common for most execution time control implementations is that they charge the running task the execution time of interrupt handlers. When the authors at NTNU implemented Ada 2005 execution time control for our AVR32 version of the GNAT bare-board run-time environment [7], separate execution

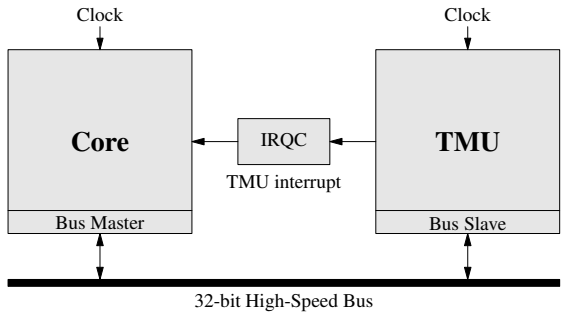

Fig. 1. Initial design with the CPU core and TMU connected to a high-speed bus.

time measurement for each interrupt level was added [5, 9]. This improved accuracy of execution time measurement and allowed execution time control for interrupts. This solution was presented at IRTAW 14 [8]. Another solution presented by the developers of MaRTE measures the combined execution time of interrupt handling [15]. Following the recommendations of the workshop [14, 20], the draft for the Ada 2012 standard [12] includes both combined execution time measurement and separate for each interrupt. The new features have been implemented by the authors [10] [1].

Performance testing has shown a significant overhead to context switches and interrupt handling, introduced by implementing execution time control [9, 10]. This motivated the authors to design a hardware Time Management Unit (TMU) to reduce the overhead [6]. The design has been implemented for Atmel AVR32 UC3 by a master student at NTNU in cooperation with Atmel Norway [18].

In the following there is a description of the TMU design and its UC3 implementation. Then follows a description of the Ada 2012 execution time control features, and our implementation of this without the TMU. After this it is shown how our implementation is modified for using the TMU and performance test results are given. Finally there is a discussion on the TMU design, the performance test results, and the portability of the solution.

2 The Time Management Unit (TMU)

The TMU was designed as a memory-mapped device accessible through a high-speed bus as shown in Figure 1. In addition to bus interface the TMU has a clock signal as input that need not be the same as the clock used by the core. The TMU generates an interrupt signal that will usually be routed to the core through an interrupt controller.

Internally the TMU has a 64-bit COUNT register that is incremented on every positive edge of the clock signal. After COUNT is incremented it is compared with the 64-bit COMPARE register. If COUNT $\geq$ COMPARE then the

[1] Code available at `http://github.com/gregerts/GNATforAVR32`

Table 1. User interface of the TMU.

Offset	Register	Description
0x00	CTRL	Control register
0x04	MODE	Mode register
0x08	SR	Status register
0x0c	SCR	Status clear register
0x10	IER	Interrupt enable register
0x14	IDR	Interrupt disable register
0x18	IMR	Interrupt mask register
0x1c	COMPARE_HI	Compare register
0x20	COMPARE_LO	
0x24	COUNT_HI	Count register
0x28	COUNT_LO	
0x2c	SWAP_COMPARE_HI	Swap compare register
0x30	SWAP_COMPARE_LO	
0x34	SWAP_COUNT_HI	Swap count register
0x38	SWAP_COUNT_LO	

interrupt signal is asserted. In order to atomically swap a new set of COUNT / COMPARE values with the current, two swap registers are provided. The registers are swapped when the final word of the swap registers is written, and the previous values of COUNT and COMPARE can be read back. The swap registers allow for simple and efficient change of execution time clocks.

The COUNT and COMPARE register may also be accessed directly. When reading the high-word of the registers, the low-word is stored in an internal 32-bit buffer, and this buffered value is returned when the low-word is later read. Similarly, the high-word value is buffered when writing the high-word of COUNT and COMPARE. The whole register is updated when the low-word subsequently is written. Due to the buffering care must be taken not to interleave writing and reading of COUNT. If available it is recommended to use double-word load / store instructions so that registers are read and written atomically.

2.1 UC3 implementation of TMU

The Atmel AVR32 [1] is a 32-bit RISC architecture optimized for code density and power efficiency. The AVR32 has four interrupt levels, and a number of exceptions. The UC3 is the second implementation of the architecture [2], intended for embedded control applications. It has a three-stage pipeline integrated with an internal SRAM allowing deterministic, single-cycle memory access.

When TMU was implemented for the UC3 some technical changes were needed [18]. The unit was moved from the high-speed bus to the peripheral bus to ease the implementation, and the clock signal driving the TMU was bound to the clock of the peripheral bus to allow a synchronous design. To make the TMU more like other UC3 peripherals and usable for a wider range of purposes, several registers were added as seen in Table 1. The control register allows enabling and

disabling the TMU. It is disabled by default to save power. Even though the 64-bit COUNT register is not expected to overflow with the intended usage, an overflow interrupt was added to allow for other usages. Also interrupt control registers were added following the pattern of existing UC3 peripherals.

3 Ada 2012 execution time control

The package Ada.Execution_Time defines the type CPU_Time and the function Clock for execution time measurement of tasks [12]. The execution time of a task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf [12]. For Ada 2005 it was implementation defined which task, if any, was charged the execution time used by interrupt handlers and run-time services on behalf of the system. Ada 2012 has the ability to account for the total or separate execution time of interrupts handlers. If supported the function Clock_For_Interrupts returns the total execution time of interrupt handlers since system start-up. The child package Interrupts is new for Ada 2012, and has a function Clock that returns the execution time spent handling the given Interrupt_Id since start-up if supported.

The child package Timers defines the tagged type Timer used for detecting execution time overruns for a single task. The type Timer_Handler identifies a protected procedure to be executed when the timer *expires*. Handlers are set to expire at an absolute or relative execution time using two overloading Set_Handler procedures, and may be cancelled using the procedure Cancel_Handler. To allow execution time control for interrupts in the same way as for tasks we have added a child package Interrupts.Timers. It defines the tagged type Interrupt_Timer that inherits Timer and its operations [10]. This package is not in Ada 2012, but should in the authors opinion be added to the next revision of the language.

4 Implementation without TMU

We have modified our earlier implementation of Ada 2012 execution time control [10] to use the TMU. To understand the changes and the overall design of the system a brief description of this implementation is needed.

4.1 Design

The real-time clock (RTC) and execution time clocks (ETCs) are quite similar in functionality: both clocks support high accuracy measurement of the monotonic passing of time since an epoch, and both support calling a protected handler when a given timeout time is reached. The main difference is that the RTC is always active, while an ETC is active only when its corresponding task or interrupt is executed. Our design takes advantage of this by having a single implementation of clocks and alarms in the internal package System.BB.Time.

In this package the type Time represents the passing of time since the epoch as a 64-bit modular integer, and Time_Span represents time differences as a 64-bit

integer with range from -2^{63} to $2^{63}-1$. The limited private types representing clocks and alarms are defined as shown in Listing 1, and there are access types Clock_Id and Alarm_Id for these. The package also defines public routines for clock and alarm operations, and procedures used by the run-time environment for changing the active execution time clock. Note that the alarm type is also used internally for task wake-up.

4.2 Hardware timer

The 32-bit COUNT / COMPARE system registers of the AVR32 are used both for the RTC and execution time clocks in the implementation without TMU. The COUNT register is reset to zero at system start-up and is incremented by one every CPU clock cycle. The COMPARE interrupt is triggered when COUNT equals COMPARE, and cleared when COMPARE is written. The interrupt is disabled when COMPARE is zero, which is the reset value of the register.

The package CPU_Primitives provides three hardware timer operations for the COUNT / COMPARE registers. A snap-shot value of COUNT is returned by the function Get_Count. The procedure Adjust_Compare sets COMPARE according to the argument C, while making sure no interrupt is lost. If C is less than COUNT, an interrupt will be pending immediately after leaving the procedure. The procedure Reset_Count sets COUNT to zero and returns the previous COUNT value c_p in one atomic operation. The COMPARE register is not altered by the reset procedure and has to be updated with a call to Adjust_Compare if needed.

4.3 Clock management

The package body has Clock_Descriptors for the RTC, interrupt clocks and the internal idle clock. Threads have a Clock_Descriptor stored in the Thread_Descriptor type. After initialization there are two active clocks: the RTC that is always active and the ETC that points to the clock of the running thread, that of the interrupt being handled or to the idle clock. The ETC is changed by the procedure Update_ETC as a result of a context switch, interrupt handling, or system idling.

The low-level interrupt handler calls Enter_Interrupt prior to calling the interrupt handler. This procedure activates the interrupt clock found in a look-up table as the new ETC. A stack is used to keep track of nested interrupts. After the interrupt has been handled the procedure Leave_Interrupt is called, and the interrupted clock is popped from the stack and reactivated.

The run-time environment has no idle thread. Instead the thread τ_a that finds the ready queue empty when leaving the kernel, enters an idle-loop waiting for any thread to be made runnable by an interrupt. Prior to entering the idle loop a call to Enter_Idle activates the idle clock as the ETC. If τ_a is made runnable it calls Leave_Idle to reactivate its clock. Also a context switch may change to a new running thread τ_b. When τ_a resumes execution the idle clock has to be activated by the context switch. Therefore the Thread_Descriptor has a field Active_Clock that points either to the task's own clock, or the idle clock if the task is executing the idle loop. Only one thread at a time will enter the idle loop.

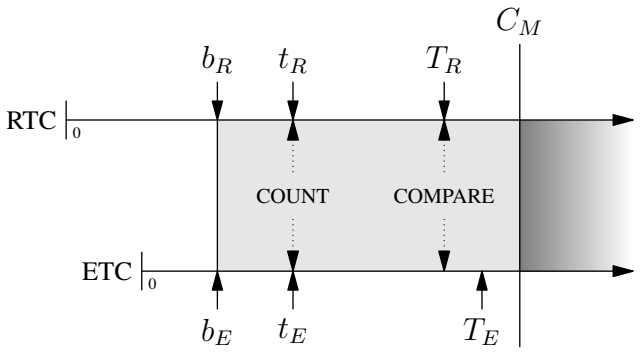

Fig. 2. Relation between the RTC and ETC, and the hardware timer registers. The base time of the two clocks are aligned in the figure.

4.4 Measuring time

The use of the hardware timer is tick-less and does not require a periodic clock overflow interrupt. Instead COUNT is reset using Reset_Count when the ETC is changed by Update_ETC, and the Base_Time of the RTC and the old ETC is incremented with the previous COUNT value c_p. By doing this the same hardware timer may be used for both the RTC and the ETC as seen in Figure 2.

The elapsed time of a clock t since the epoch is retrieved by the function Elapsed_Time, and is computed from the base time b and the COUNT register value:

$$t = \begin{cases} b + \text{COUNT} & \text{if clock is active} \\ b & \text{else} \end{cases}$$

An interrupt may occur while within Elapsed_Time. This would reset COUNT and update the base time. To avoid an invalid result there is a check after reading COUNT to see if the base time has been updated, in which case the updated base time will be returned as the elapsed time.

4.5 Setting the hardware timer

The COMPARE register is adjusted after updating ETC and after the first alarm of an active clock is changed. This is done by calling the procedure Update_Compare. As seen in Figure 2, the value C given to Adjust_Compare is shortest remaining time until timeout T for the RTC and ETC. However, this value is never greater than the maximal COMPARE value C_M to avoid COUNT from overflowing. This safety region is marked with a darker shade in Figure 2. The interrupt will be pending within this region and COUNT is reset when it is handled, preventing overflow. The COMPARE interrupt handler will simply ignore this "false" interrupt. A large safety region of 2^{31} cycles is used to provide ample time for the interrupt to be handled.

4.6 Alarms

Each clock has a queue of pending alarms managed as a linked list and sorted in ascending order after the Timeout of the alarms. In the case of equal Timeout values alarms are queued in FIFO order. To avoid the special condition of an empty queue, there is a sentinel alarm with timeout at Time'Last that is always present at the end of the queue. The constant Time_Last seen by the user is set to Time'Last − 1 so that the sentinel is always last. This avoids an additional check when searching the queue. One sentinel alarm without handler is shared between all clocks to save memory. The procedures Set and Cancel both search the queue from the start for the place to insert or remove an alarm, and call Update_Compare if the first alarm in the queue of an active clock is changed.

4.7 Interrupt handler

The COMPARE interrupt handler has the highest interrupt priority. The handler calls the procedure Alarm_Wrapper first for the RTC and then for the interrupted ETC on top of the stack. At this point the active ETC is that of the COMPARE interrupt itself, for which no alarms are allowed, so only the interrupt ETC on top of the stack or the RTC may be the cause of the interrupt. As the wrapper is called for both clocks there is no need to check which one caused the interrupt. The alarm wrapper removes all alarms with timeout less or equal to the base time of the clock from the head of alarm queue one at the time, clears the alarm and calls the handler with the data as argument.

5 Modifications for using the TMU

The package specification of System.BB.Time was not altered when using the TMU. However, the routines interfacing with hardware and the COMPARE interrupt handler has to be updated. Also, an interrupt handler for the TMU has to be added, and an interface to the TMU in the package System.BB.Peripherals.

5.1 TMU interface

A modular integer type TMU_Interval is defined to represent the 64-bit timer values of the TMU, and the memory-mapped interface is defined in the child package Registers. Public routines were added to read the COUNT register, set the COMPARE register and perform the swap operation. The TMU is configured and enabled as a part of the peripheral initialization.

5.2 Hardware timer usage

The updates needed to use two separate hardware timers are shown in Listing 2. The function Elapsed_Time is updated to use the TMU if the given clock is the ETC. No check for interruption is needed for this as the double-word read cannot

be interrupted. If the clock is neither the RTC nor the ETC, it is not active and its elapsed time equals the Base_Time. The procedure Update_Compare updates the correct hardware timer if the given clock is active. The RTC is no longer updated by Update_ETC the TMU swap operation simplifies the procedure.

5.3 Interrupt handlers

The COMPARE interrupt handler now only handles alarms for the RTC. As the RTC is no longer updated when changing the ETC, we reset COUNT and update the base time of the RTC first in this handler. The flag Defer_Updates is set while calling Alarm_Wrapper to avoid needless COMPARE updates. Notice the use of the flag in Update_Compare in Listing 2. The COMPARE register is updated after the user handlers are called.

The TMU interrupt handler only calls Alarm_Wrapper for the interrupted clock on top of the stack. This as before this interrupted clock is the only possible source of the interrupt. Updates to the TMU need not be deferred as the clock is not active.

5.4 Context switch

The context switch routine now changes the active ETC directly as shown in Listing 3. First the Base_Time and First_Alarm . Timeout of the new running threads Active_Clock are loaded. Then a TMU swap operation is initiated using the multiple store instruction of the AVR32 architecture. Notice that the registers are stored in reverse order and therefore the high-word is stored before the low-word. After the swap operation the COUNT value of the previous ETC is read back and stored as its Base_Time. Finally, the ETC is updated to the new active clock.

6 Performance testing

To evaluate the implementation of execution time control using the TMU, we execute performance tests and compare the results with those for the implementation using the COUNT / COMPARE registers [10]. We also compare to results without execution time control to find the absolute overhead caused by implementing this feature. The implementations are referred to as TMU-ETC, CC-ETC, and N-ETC respectively.

The testing of the TMU was done by simulation as it has not yet been included in a produced UC3 chip. However, since the syntheziable RTL code of the UC3 was used the results are the same as if obtained on hardware. The run-time environment and test programs are compiled and linked to an ELF file as normal, and no special code or libraries were needed to execute on the simulator. The test programs are the same with exception of the non-simulated tests sending data over the USART line, while the simulated store data in memory to be read directly using the simulator. Some updates to the run-time environment were needed as the simulated microcontroller is of version UC3L, while the earlier tests were for the UC3A [10]. These differences do not affect the test results.

Table 2. Performance test results in CPU cycles.

Test	Implementation		
	TMU-ETC	CC-ETC	N-ETC
Reading the RTC	43	51	41
Reading the ETC	47	56	–
Context switch	529	602	471
Interrupt handler	294	324	204
Timing event	369	381	270
Interruption cost	244	295	–

6.1 Reading the RTC and ETC

The purpose of this test is to find the overhead of reading the RTC and the active execution time clock (ETC). This is important as this overhead affects most of the later test results. The test is done by a task reading the RTC twice, and then its own execution time clock twice before the results are stored in memory. After this the task is delayed for a short while so that COUNT is reset and there will be no interrupts while reading the clocks. The overhead is calculated as the difference between the two clock values read.

Due to the deterministic nature of the UC3 microcontroller and the simplicity of the test program, all samples for all implementations were of the exact same value for this test. As seen from Table 2, the time to read the RTC and the ETC is reduced by 7 and 9 clock cycles for TMU-ETC compared to CC-ETC, a reduction of 14% and 16% respectively. The overhead of reading the RTC for TMU-ETC is only 2 cycles or 5% more than for N-ETC.

6.2 Context switch overhead

The purpose of this test is to find the overhead to context switches by changing the execution time clock. We test without an alarm being set for the clock as the overhead is found to be the same regardless of alarm status. The test is done by task τ_a releasing a higher priority task τ_b that is blocked on a protected entry. The release time is read by the protected procedure opening the entry and is returned to τ_b. After being released τ_b reads the clock and stores the data in memory before it blocks again and the test is repeated. The time between the two clock readings thus include finishing the protected procedure, executing the entry by proxy on behalf of τ_b, leaving the protected object, the context switch to τ_b and retrieving the results of the entry call.

For the same reasons as the previous test, all samples were of the same value for this test. If we subtract the overhead of reading the RTC from the results in Table 2, it can be inferred that TMU-ETC has a context switch overhead caused by execution time control of 56 clock cycles compared to 121 for CC-ETC. This is an reduction of 65 cycles, or 54%.

6.3 Interrupt handler overhead

The purpose of this test is to find the interrupt handler overhead caused by implementing execution time control for interrupts. The test is done by using the 16-bit Timer / Counter (TC) peripheral unit of the UC3. The TC is set up to generate interrupts at regular intervals each time its counter is reset. The counter value is read by the interrupt handler and stored in memory. This provides a good measurement of the overhead from the interrupt line is asserted to interrupt handler is called. The sample values are multiplied with the clock division factor used by the TC to get the time in CPU cycles.

As before, all samples were of the same value for this test. As seen from Table 2, the overhead caused by execution time control is reduced from 120 clock cycles for CC-ETC to 90 clock cycles for TMU-ETC. This is a reduction of 30 clock cycles or 25%.

6.4 Timing event overhead

The system is required to document the overhead of handling timing event occurrences. While not related to the execution time control, this overhead is expected to be changed by our implementation using the TMU and had to be found. The test program has a single timing event that is programmed to occur with random intervals between 1 and 3 milliseconds. When the handler is called the difference between the timeout and the value of the RTC is stored in memory.

As before, all samples were of the same value for this test. By subtracting the overhead of reading the RTC from the results found in Table 2, it can be inferred the timing event overhead caused by execution time control is reduced from 101 clock cycles for CC-ETC to 97 clock cycles for TMU-ETC. This is a reduction of 4 clock cycles, or 4%.

6.5 Cost to interrupted task

The execution time cost to the task being interrupted is greater than zero, as the interrupt clock is activated by the low-level interrupt handler. The purpose of this test is to find this cost. The test is done by a task τ first setting its own execution time timer to expire in 20 ms, then reading its execution time clock, busy waiting 10 millisecond and then reading this clock again. The clock values are stored in memory and the test is repeated. Only the interrupt caused by the timer can occur between the two clock readings, and it can occur only once. To find the cost we compare the difference in execution time when interrupted to when the task is not interrupted. This test is not possible for N-ETC due to the lack of execution time measurement.

For this test there was a difference of one clock cycle between the maximal and minimal sample for CC-ETC. The maximal sample value for this implementation is shown in Table 2. If we subtract the overhead of reading the execution time clock from the results found in Table 2, it can be inferred that the cost to the interrupted task is 239 clock cycles for CC-ETC and 197 clock cycles for TMU-ETC. This is a reduction of 42 clock cycles or 21%.

Table 3. Performance improvements with TMU.

Test	Improvement	
	CPU cycles	Reduction (%)
Reading the RTC	7	14
Reading the ETC	9	16
Context switch	65	54
Interrupt handler	30	25
Timing event	4	4
Interruption cost	42	21

7 Discussion

7.1 Performance improvements

Testing showed that the TMU reduced the overhead and therefore improves the performance of the system. However, as seen from the overview in Table 3 some improvements were more significant than others. The overhead of handling timing events is hardly reduced at all. This is explained by the RTC now being reset before calling the handler in addition to the change of ETC. The implementation CC-ETC does both in one operation when the ETC is updated and is therefore almost as efficient as TMU-ETC. Also, while the relative overhead reduction for reading clocks is good, the absolute reduction is only a few clock cycles and does not affect the system performance much.

There is a noticeable improvement in interrupt handling latency. This is caused by the reduced execution time of Update_ETC using the TMU swap operation. Related to this is the improvement in cost to the interrupted task, that also has a noticeable improvement. Further improvements could be achieved if the swap operation was moved to the assembler part of the low-level interrupt handler. Yet, this has to be weighted against the added complexity and reduced maintainability by moving functionality from Ada to assembler.

The best improvement is for the context switch. This was expected as a complex procedure was replaced by the few assembly code lines seen in Listing 3. Combined with the general speed-up of changing clocks for the TMU, this more than halves the overhead introduced by execution time control compared to the earlier implementation. In systems with frequent context switches this should give a noticeable performance improvement.

7.2 Modifications of run-time environment

As the package specification of System.BB.Time is unchanged the modifications for using the TMU are isolated to the package body, the context switch routine, the package System.BB.Peripherals and its child package Registers. The modifications within the package body of System.BB.Time are limited to the low-level parts

interfacing with hardware clocks. The high-level parts concerned with alarms and managing clocks are unchanged.

The body of System.BB.Time has two logical code lines *less* when using the TMU. For the peripheral packages 50 logical code lines were added for interfacing with the TMU, whereof only 8 are statements. For the context switch only 8 additional instructions were needed, all simple load, store or move instructions. In essence the complexity of the run-time environment as a whole is unchanged when using the TMU.

7.3 TMU design, implementation and portability

Our TMU is a simple, yet highly efficient, hardware mechanism for implementing execution time control that leaves the policy entirely for the software. This simplifies the hardware implementation and is also more flexible as the usage of the TMU is decided by software. In contrast, an earlier design [17] implemented for the LEON 2 architecture, changed clocks automatically before the processor started handling an interrupt [4]. This design also supported blocking the interrupt in hardware after the deferrable server pattern. While the benefit of this design is zero overhead to interrupt handling, it is costly to implement and also limits the choice of execution time control policy to one predefined in hardware.

When the TMU was implemented for UC3 some minor changes were needed for easing the implementation, and making the unit more usable for a wider range of applications [18]. The only noticeable change for our implementation of execution time control is that the TMU was moved from the high-speed bus to the peripheral bus. This eased the hardware implementation and reduced the cost in number of gates, but also increases the access latency for the registers. However, the UC3 allows the creation of a CPU local bus to the TMU [18]. If implemented this would provide single-cycle access to the TMU registers.

Since the TMU is designed as a simple memory-mapped device without any special system requirements, it should be portable to other architectures. In essence only the parts needed for interfacing with the memory-mapped bus need to be changed, and the TMU can be integrated with the system-on-chip by connecting the bus and interrupt line. In contrast the earlier TMU design modified the interrupt lines and is much harder to implement on existing architectures.

8 Conclusion

The careful design of our Time Management Unit (TMU) with 64-bit time measurement and the special swap operation, allowed us to develop a highly efficient implementation of Ada 2012 execution time control for the Atmel AVR32 UC3 microcontroller series. Only minor changes were needed to our earlier implementation in order to use the TMU. Performance testing with the UC3 has shown that the TMU gives a significant reduction of the overhead for context switches and interrupt handling, and also reduces the execution time cost for the interrupted task. This makes real-time applications taking advantage of execution time control more efficient and analyzable.

Acknowledgments

Thanks to Atmel Norway and Frode Sundal for facilitating the simulation work. Special thanks to Martin Olsson for the support during the simulation process.

References

1. Atmel Corporation: AVR32 - Architecture Document (November 2007), http://atmel.com/dyn/resources/prod_documents/doc32000.pdf
2. Atmel Corporation: AVR32UC3 - Technical Reference Manual (March 2010), http://atmel.com/dyn/resources/prod_documents/doc32002.pdf
3. Burns, A., Wellings, A.: Programming execution-time servers in Ada 2005. In: Proc. 27th IEEE International Real-Time Systems Symposium RTSS '06. pp. 47–56 (Dec 2006)
4. Forsman, B.: A Time Management Unit (TMU) for Real-Time Systems. Master's thesis, Norwegian University of Science and Technology (NTNU) (2008)
5. Gregertsen, K.N.: Execution Time Management for AVR32 Ravenscar. Master's thesis, Norwegian University of Science and Technology (NTNU) (2008)
6. Gregertsen, K.N., Skavhaug, A.: Functional specification for a Time Management Unit, presented at SAFECOMP 2010.
7. Gregertsen, K.N., Skavhaug, A.: An efficient and deterministic multi-tasking runtime environment for Ada and the Ravenscar profile on the Atmel AVR32 UC3 microcontroller. In: Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09. pp. 1572–1575 (April 2009)
8. Gregertsen, K.N., Skavhaug, A.: Execution-time control for interrupt handling. Ada Lett. 30 (2010)
9. Gregertsen, K.N., Skavhaug, A.: Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture. Journal of Systems Architecture 56, 509–522 (2010)
10. Gregertsen, K.N., Skavhaug, A.: Implementation and usage of the new Ada 2012 execution-time control features. Ada User Journal 32(4), 265–275 (December 2011)
11. ISO/IEC: Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1., http://www.adaic.com/standards/05rm/html/RM-TOC.html
12. ISO/IEC: Ada Reference Manual - ISO/IEC 8652:201x(E) (Draft 15), http://www.ada-auth.org/standards/ada12.html
13. Krishna, C.M., Shin, K.G.: Real-Time Systems. McGraw-Hill International Edition (1997)
14. Michell, S., Real, J.: Conclusions of the 14th International Real-Time Ada Workshop. Ada Lett. 30 (2010)
15. Rivas, M.A., Harbour, M.G.: Execution time monitoring and interrupt handlers: position statement. Ada Lett. 30 (2010)
16. Santos, O.M., Wellings, A.: Cost enforcement in the real-time specification for Java. Real-Time Systems 37(2), 139–179 (2007)
17. Skinnemoen, H., Skavhaug, A.: Hardware support for on-line execution time limiting of tasks in a low-power environment. In: EUROMICRO / DSD Work in progress session. Linz: Institute of system science, Johannes Kepler University (2003)
18. Søvik, S.J.: Hardware implementation of a Time Management Unit. Master's thesis, NTNU (2010)

19. The Open Group: The Open Group base specifications issue 6, IEEE Std 1003.1. Web: `http://www.opengroup.org/onlinepubs/000095399/`, `http://www.opengroup.org/onlinepubs/000095399/`
20. Vardanega, T., Harbour, M.G., Pinho, L.M.: Session summary: language and distribution issues. Ada Lett. 30 (2010)
21. Wellings, A., Burns, A.: Ada-Europe 2007, chap. Real-Time Utilities for Ada 2005, pp. 1–14. Springer Berlin / Heidelberg (2007)
22. Wilhelm, R., et al.: The worst-case execution-time problem—overview of methods and survey of tools. Trans. on Embedded Computing Sys. 7(3), 1–53 (2008)

Listing 1. Definition of clocks and alarms

```
type Clock_Descriptor is
   record

      Base_Time : Time;
      -- Base time of clock

      First_Alarm : Alarm_Id;
      -- First alarm of clock

      Capacity : Natural;
      -- Remaining alarm capacity

   end record;

type Alarm_Descriptor is
   record

      Timeout : Time;
      -- Timeout of alarm when set

      Clock : Clock_Id;
      -- Clock of this alarm

      Handler : Alarm_Handler;
      -- Handler called when alarm expires

      Data : System.Address;
      -- Argument when calling handler

      Next : Alarm_Id;
      -- Next alarm in queue when set

   end record;
```

Listing 2. Updates to use TMU as hardware timer.

```
function Elapsed_Time (Clock : not null Clock_Id) return Time is
begin

   if Clock = RTC'Access then
      return T : Time := Clock.Base_Time do
         T := T + Time (CPU.Get_Count);
         CPU.Barrier;
         if T < Clock.Base_Time then
            T := Clock.Base_Time;
         end if;
      end return;
   elsif Clock = ETC then
      return Time (Peripherals.Get_Count);
   else
      return Clock.Base_Time;
   end if;

end Elapsed_Time;

procedure Update_Compare (Clock : Clock_Id) is
   T : constant Time := Clock.First_Alarm.Timeout;
begin

   if Clock = RTC'Access and then not Defer_Updates then
      declare
         R : constant Time := T - Time'Min (T, Clock.Base_Time);
      begin
         CPU.Adjust_Compare (CPU.Word (Time'Min (R, Max_Compare)));
      end;
   elsif Clock = ETC then
      Peripherals.Set_Compare (Peripherals.TMU_Interval (T));
   end if;

end Update_Compare;

procedure Update_ETC (Clock : Clock_Id) is
   use Peripherals;
begin
   pragma Assert (Clock /= null);

   Swap_Context (TMU_Interval (Clock.First_Alarm.Timeout),
                 TMU_Interval (Clock.Base_Time),
                 TMU_Interval (ETC.Base_Time));

   ETC := Clock;

end Update_ETC;
```

Listing 3. Context switch routine

```
/* Store address of running thread in r9 */
lda.w   r8, running_thread
ld.w    r9, r8

/* Add size of context */
sub     r9, -CONTEXT_SIZE

/* Save CPU context of running thread */
stm     --r9, r0,r1,r2,r3,r4,r5,r6,r7,sp, lr
mfsr    r0, SYSREG_SR
st.w    --r9, r0

/* Store address of first thread in r1 */
lda.w   r1, first_thread
ld.w    r9, r1

/* First thread is now also running thread */
st.w    r8, r9

/* Load Active_Clock of first_thread */
ld.w    r0, r9[THREAD_ACTIVE_CLOCK_OFFSET]

/* Load First_Alarm.Timeout and Base_Time */
ld.w    r1, r0[CLOCK_FIRST_ALARM_OFFSET]
ld.d    r4, r1[ALARM_TIMEOUT_OFFSET]
ld.d    r2, r0[CLOCK_BASE_TIME_OFFSET]

/* Do TMU swap operation */
mov     r1, TMU_ADDRESS + TMU_SWAP_OFFSET
stm     r1, r2-r5
ld.d    r4, r1[8]

/* Load ETC address */
lda.w   r1, system__bb__time__etc

/* Load current ETC and store its Base_Time */
ld.w    r2, r1
st.d    r2[CLOCK_BASE_TIME_OFFSET], r4

/* Active_Clock of first_thread is now ETC */
st.w    r1, r0

/* Load CPU context of first thread */
ld.w    r0, r9++
mtsr    SYSREG_SR, r0
sub     pc, -2
ldm     r9++, r0,r1,r2,r3,r4,r5,r6,r7,sp,pc
```

Appendix B

Presentations at IRTAW-14

In this appendix are the slides from the presentations by the author at the 14th International Real-Time Ada Workshop (IRTAW-14). The workshop was held in Portovenere, Italy, from the 7 to 9 of October 2009. The slides are included as they were presented at the workshop.

NTNU
Norwegian University of
Science and Technology

Execution-time control for interrupt handling

Kristoffer Nyborg Gregertsen, Amund Skavhaug
Department of Engineering Cybernetics
14th International Real-Time Ada Workshop

Proposed additions

- Propose adding support for:
  - Execution-time measurement for interrupt priorities
  - Execution-time timers for interrupt priorities
- Defines one pseudo server task for each interrupt priority handling all interrupts of that priority
- Have execution-time clock and timer for these pseudo tasks
- Based on my master thesis at NTNU in spring 2008
- Implemented on GNAT bare-board Ravenscar run-time environment for the Atmel AVR32 UC3 microcontroller series

NTNU
Norwegian University of
Science and Technology

3

Execution_Time

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is

   type CPU_Time is private;

   ...

   function Clock (T : Ada.Task_Identification.Task_Id
                     := Ada.Task_Identification.Current_Task) return CPU_Time;

   function Interrupt_Clock (I : System.Interrupt_Priority) return CPU_Time;

   ...

private
   ...
end Ada.Execution_Time;
```

NTNU
Norwegian University of
Science and Technology

4

Timers

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Task_Identification; use Ada.Task_Identification;
with System;
package Ada.Execution_Time.Timers is

   type Timer (T : not null access constant Task_Id) is tagged limited private;

   ...

   Pseudo_Task_Id : aliased constant Task_Id := Null_Task_Id;

   type Interrupt_Timer (I : System.Interrupt_Priority)
      is new Timer (Pseudo_Task_Id'Access) with private;

   ...

private
   ...
end Ada.Execution_Time.Timers;
```

NTNU
Norwegian University of
Science and Technology

5

AVR32 implementation

- Use COUNT / COMPARE registers
- Very low overhead to reprogram timer (50 CPU cycles)
- Change active timer from low-level interrupt handler
- Manages timers in a stack to allow nested interrupts
- At most one timer per task and interrupt priority
- No timer for highest interrupt priority
- Low cost for interrupt task (150 CPU cycles)
- Also account for execution-by-proxy of entries

NTNU
Norwegian University of
Science and Technology

6

Interrupt state and server

```
package Interrupt_States is

   type Interrupt_State is limited interface;

   procedure Handler (S : in out Interrupt_State) is abstract;
   procedure Enable  (S : in out Interrupt_State) is abstract;
   procedure Disable (S : in out Interrupt_State) is abstract;

   type Any_Interrupt_State is access all Interrupt_State'Class;

end Interrupt_States;

package Interrupt_Handlers is

   protected type Interrupt_Handler
     (Id  : Interrupt_Id;
      Pri : Interrupt_Priority;
      S   : not null Any_Interrupt_State) is
     pragma Interrupt_Priority (Pri);
   private
      procedure Handler;
      pragma Attach_Handler (Handler, Id);
   end Interrupt_Handler;

end Interrupt_Handlers;
```

NTNU
Norwegian University of
Science and Technology

7

Interrupt server

```
package Interrupt_Servers is

   type Interrupt_Server_Parameters is
      record
         Pri : Interrupt_Priority;
         Budget : Time_Span;
         Period : Time_Span;
      end record;

   type Interrupt_Server is limited interface;

   procedure Register
     (S : in out Interrupt_Server;
      I : Any_Interrupt_State) is abstract;

   type Any_Interrupt_Server is access all Interrupt_Server;

end Interrupt_Servers;
```


8

Deferrable interrupt server

```
protected type Deferrable_Interrupt_Server
  (Param : access Interrupt_Server_Parameters)
is new Interrupt_Server with

   procedure Register (I : Any_Interrupt_State);
   pragma Interrupt_Priority (Any_Priority'Last);

private

   procedure Replenish (TE : in out Timing_Event);
   procedure Overran (TM : in out Timer);

   Replenish_Event : Timing_Event;
   Execution_Timer : access Interrupt_Timer;

   Next : Time;
   Disabled : Boolean := True;
   Registered : Natural := 0;
   States : State_Array;

end Deferrable_Interrupt_Server;
```

9

Deferrable interrupt server

```
procedure Replenish (TE : in out Timing_Event) is
begin
   Execution_Timer.Set_Handler (Param.Budget, Overran'Access);
   if Disabled then
      Disabled := False;
      for I in 1 .. Registered loop
         States (I).Enable;
      end loop;
   end if;
   Next := Next + Param.Period;
   TE.Set_Handler (Next, Replenish'Access);
end Replenish;

procedure Overran (TM : in out Timer) is
begin
   if not Disabled then
      Disabled := True;
      for I in 1 .. Registered loop
         States (I).Disable;
      end loop;
   end if;
   -- Set fallback handler for TM
end Overran;
```

NTNU
Norwegian University of
Science and Technology

10

Benefits

1. Improves accuracy of execution-time measurement for tasks allowing tighter task budgets
2. Possible to control execution-time spent handling interrupts making it possible to protect the system from burst of interrupts

NTNU
Norwegian University of
Science and Technology

11

Multiprocessor issues

- Implemented and tested on uni-core AVR32
- For multiprocessors need to define pseudo server tasks for each processor that may handle interrupts
- An alternative is to use Interrupt_Id instead of priority:
  - Will also work for multiprocessors
  - Implemented (took 20 min to change from using priorities)
  - Needs more memory (no big deal for larger systems)
- Used priorities as this is most efficient for uni-processors

NTNU
Norwegian University of
Science and Technology

12

Other issues

- Definition of Interrupt_Timer may be more elegant
- Need to have an Task_Id since inheriting Timer
- Maybe define abstract type Root_Timer that is inherited by both Timer and Interrupt_Timer
- This would remove need for (nasty) Pseudo_Task_Id
- May be better to have new functionality in child packages?

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Execution-time control for interrupt handling

Kristoffer Nyborg Gregertsen

IRTAW-14 – Portovenere

Proposed API

- Defines one execution-time clock for each Interrupt_ID
- Returns time spent handling that interrupt
- May also have timers associated with these clocks
- Interrupt timer inherits ordinary (task) timer
- Has Null_Task_Id as discriminant (need to be aliased)

NTNU
Norwegian University of
Science and Technology

3

Ada.Execution_Time.Interrupts

```
with Ada.Interrupts;

package Ada.Execution_Time.Interrupts is

   function Clock (I : Ada.Interrupts.Interrupt_ID) return CPU_Time;

private
   ...
end Ada.Execution_Time.Interrupts;
```

NTNU
Norwegian University of
Science and Technology

4

Ada.Execution_Time.Timers.Interrupts

```
with Ada.Interrupts;

package Ada.Execution_Time.Timers.Interrupts is

   type Timer (I : Ada.Interrupts.Interrupt_ID)
      is new Ada.Execution_Time.Timers.Timer
     (Ada.Task_Identification.Null_Task_Id'Access) with private;

private
   ...
end Ada.Execution_Time.Timers.Interrupts;
```

NTNU
Norwegian University of
Science and Technology

5

Rationale

— Having a Task_Id for pseudo tasks may seem elegant, but:
- These are not normal Ada tasks
- Cannot do task operations on them (like Set_Priority)
- Need checks for these special Task_Ids *everywhere*

— Using Interrupt_ID for identifying the pseudo task:
- Works with multiprocessors
- Only need to allocate data when registering a handler

— Having one pseudo task for each Interrupt_Priority works well on uni-processors, but on multiprocessors need to define these for each processor

NTNU
Norwegian University of
Science and Technology

6

Why?

— Not charging interrupted tasks the execution-time of interrupt handlers improves accuracy of execution-time measurement

— If tasks are not to be charged the execution-time of interrupts then another entity should be

— If it makes sense to monitor the execution-time of tasks, why not also monitor the execution-time of interrupts?

— Interrupt timers allows for protecting the system from bursts of interrupts caused by hardware error (or other reasons)

— You *may* count occurrences using Ada 2005, but you cannot know how much execution-time is spent handling them

NTNU
Norwegian University of
Science and Technology

Appendix C

GNATforAVR32

This appendix describes how the sources for GNATforAVR32 are obtained and how to configure and build the cross-development environment under GNU / Linux. The most important files of the run-time environment are also described.

The system is under active development by the author and the source code is available for download and online browsing through the github web-service at `https://github.com/gregerts/GNATforAVR32`.

Since Git is used for distributed version control others may easily check out different branches, contribute to the project, or even fork the code to start their own GNAT run-time environment project.

The instructions in this appendix have only been tested on the Debian and Ubuntu distributions. Basic knowledge of GCC and the GNU / Linux environment is assumed.

C.1 Installing GNAT on the host machine

In order to to build the cross-compiler for AVR32, one first needs to install the GNAT 2010 GPL compiler from AdaCore available for download at `http://libre.adacore.com`. The installation of the compiler is automatic and straightforward using the provided install-script.

After the installation update the path and test the compiler. If the installed compiler fails to find `crt0.o` there installation has failed to detect the standard C-library on

the machine due to different naming conventions. This may work to solve this problem:

```
cd /lib
sudo mkdir x86_64-pc-linux-gnu
cd x86_64-pc-linux-gnu
sudo ln -s ../x86_64-linux-gnu 4.3.6
cd /usr/lib
sudo mkdir x86_64-pc-linux-gnu
cd x86_64-pc-linux-gnu
sudo ln -s ../x86_64-linux-gnu 4.3.6
```

GNATforAVR32 currently works with GNAT 2010 that is based on GCC 4.3, as GNAT 2011 is based on GCC 4.4 while the AVR32 back-end is only available for GCC 4.3. This will not be solved until Atmel updates the AVR32 back-end to a more modern version.

C.2 Installing the GNU toolchain for AVR32

The GNU cross-toolchain for AVR32 is also needed to setup the development environment. As the toolchain available through Atmels repository is outdated it is recommended obtain the newer version by downloading and installing AVR32 Studio version 2.7 from `http://www.atmel.no/beta_ware/`.

After the ZIP-file is downloaded execute the following commands to extract the files and put the toolchain on the path:

```
cd /opt
sudo unzip /path/to/download/ase4-ide-2.7.0*.zip
sudo chgrp -R users as4e-ide
sudo ln -s $(find ./ -name avr32-gcc -printf "%h\n") as4e-ide/bin
export PATH="/opt/as4e-ide/bin:$PATH"
```

It is handy to add the export of PATH for GNAT and the AVR32 toolchain to the .bashrc file.

C.3 Obtaining and building the cross compiler

To build the cross-compiler first download and extract the patched source code:

```
mkdir avr32-gnat
cd avr32-gnat
wget folk.ntnu.no/gregerts/avr32gnat/avr32-gnat-2010.tar.bz2
tar xjf avr32-gnat-2010.tar.bz2
```

Now we may configure and build the compiler using GNAT GPL 2010:

```
mkdir obj
cd obj
../src/configure --target=avr32 \
 --enable-languages="c,ada" \
 --disable-libada --disable-threads \
 --disable-libmudflap --without-headers \
 --disable-libssp --disable-libgomp \
 --with-gnu-as --with-as=/opt/as4e-ide/bin/avr32-as \
 --with-gnu-ld --with-ld=/opt/as4e-ide/bin/avr32-ld
make
make -C gcc cross-gnattools
sudo make install
```

The compiler will take some time to build. It is recommended to find a good cup of tea or coffee – and enjoy philosophizing over a compiler compiling a compiler.

C.4 Obtaining and building the run-time environment

The run-time environment has been moved to github in order to ease the development process and allow others to participate. The source is downloaded using git:

```
git clone git://github.com/gregerts/GNATforAVR32.git
```

Now we may build the run-time environment using our newly built compiler:

```
cd GNATforAVR32/src
make
make install
```

We need to manually make links from the compiler install directory to library directory of the run-time environment in order to build ordinary applications:

```
cd /usr/local/lib/gcc/avr32/4.3.3
sudo ln -s /path/to/GNATforAVR32/build/adainclude
sudo ln -s /path/to/GNATforAVR32/build/adalib
```

If everything went well you may now compile and test a simple application for the EVK1100 evaluation board programmed with the Atmel JTAG ICE Mk. II:

```
cd GNATforAVR32/tests/timers-1
make
make install
```

s-bb.ads	Parent package of bare-board kernel.
s-bbcppr.ad{s,b}	CPU dependent operations.
s-bbinte.ad{s,b}	Interrupt registration and handling.
s-bbpara.ads	Configuration parameters for kernel.
s-bbpere.ads	Peripheral register definitions.
s-bbperi.ad{s,b}	Peripheral interface.
s-bbprot.ad{s,b}	Protection by interrupt masking and idle loop.
s-bbseou.ad{s,b}	Serial output to facilities.
s-bbthqu.ad{s,b}	Queue management for ready tasks.
s-bbthre.ad{s,b}	Thread definition and operations.
s-bbtime.ad{s,b}	Timing definition and operations.

Table C.1: *Bare-board kernel files.*

The most important files of the run-time environment for this work are those of the bare-board kernel in the package hierarchy below System.BB. Table C.1 gives a brief overview over these files.

C.5 Debugging applications

Applications may also be debugged using GDB with the JTAG ICE Mk II:

```
cd GNATforAVR32/tests/application
make
make install
make debug
```

This starts the program avr32gdbproxy. Now you may start a debugging session using avr32-gdb. In avr32-gdb (executed in the application directory with the binary 'main' as argument) type the following:

```
target extended-remote:4242
dir ../common
dir ../../src
```

You may now set breakpoints in the application and run-time environment code.

Appendix D

Additional listings

Listing D.1: *The real-time package*

```
package Ada.Real_Time is

   type Time is private;
   Time_First : constant Time;
   Time_Last : constant Time;
   Time_Unit : constant := implementation−defined−real−number;

   type Time_Span is private;
   Time_Span_First : constant Time_Span;
   Time_Span_Last : constant Time_Span;
   Time_Span_Zero : constant Time_Span;
   Time_Span_Unit : constant Time_Span;

   Tick : constant Time_Span;
   function Clock return Time;

   function "+" (Left : Time; Right : Time_Span) return Time;
    ... −− more overloaded operators

   function To_Duration (TS : Time_Span) return Duration;
   function To_Time_Span (D : Duration) return Time_Span;

   function Nanoseconds (NS : Integer) return Time_Span;
   function Microseconds (US : Integer) return Time_Span;
   function Milliseconds (MS : Integer) return Time_Span;
   function Seconds      (S  : Integer) return Time_Span;
   function Minutes      (M  : Integer) return Time_Span;

   type Seconds_Count is range implementation−defined;

   procedure Split(T : in Time; SC : out Seconds_Count; TS : out Time_Span);
   function Time_Of(SC : Seconds_Count; TS : Time_Span) return Time;

private
    ... −− not specified by the language
end Ada.Real_Time;
```

Listing D.2: *Timing events definition*

```
package Ada.Real_Time.Timing_Events is

   type Timing_Event is tagged limited private;
   type Timing_Event_Handler
         is access protected procedure (Event : in out Timing_Event);

   procedure Set_Handler (Event : in out Timing_Event;
                          At_Time : in Time;
                          Handler : in Timing_Event_Handler);
   procedure Set_Handler (Event : in out Timing_Event;
                          In_Time : in Time_Span;
                          Handler : in Timing_Event_Handler);
   function Current_Handler (Event : Timing_Event)
         return Timing_Event_Handler;
   procedure Cancel_Handler (Event   : in out Timing_Event;
                             Cancelled : out Boolean);

   function Time_Of_Event (Event : Timing_Event) return Time;

private
   ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

Listing D.3: *Execution time definition*

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;

package Ada.Execution_Time is

   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last : constant CPU_Time;
   CPU_Time_Unit : constant := implementation-defined-real-number;
   CPU_Tick : constant Time_Span;

   function Clock
      (T : Ada.Task_Identification.Task_Id
           := Ada.Task_Identification.Current_Task)
      return CPU_Time;

   function "+" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   ... -- more overloaded operations

   procedure Split
      (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

   function Time_Of (SC : Seconds_Count;
                     TS : Time_Span := Time_Span_Zero) return CPU_Time;

   Interrupt_Clocks_Supported : constant Boolean :=
      implementation-defined;

   Separate_Interrupt_Clocks_Supported : constant Boolean :=
      implementation-defined;

   function Clock_For_Interrupts return CPU_Time;

private
   ... -- not specified by the language
end Ada.Execution_Time;
```

Listing D.4: *Execution time for interrupts definition*

```
with Ada.Interrupts;

package Ada.Execution_Time.Interrupts is

    function Clock (Interrupt : Ada.Interrupts.Interrupt_Id)
        return CPU_Time;

    function Supported (Interrupt : Ada.Interrupts.Interrupt_Id)
        return Boolean;

end Ada.Execution_Time.Interrupts;
```

Listing D.5: *Execution time timers definition*

```
with System;
package Ada.Execution_Time.Timers is

   type Timer (T : not null access constant
                   Ada.Task_Identification.Task_Id) is
      tagged limited private;

   type Timer_Handler is
      access protected procedure (TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority :=
   implementation-defined;

   procedure Set_Handler (TM    : in out Timer;
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);

   procedure Set_Handler (TM    : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);

   function Current_Handler (TM : Timer) return Timer_Handler;

   procedure Cancel_Handler (TM      : in out Timer;
                             Cancelled :    out Boolean);

   function Time_Remaining (TM : Timer) return Time_Span;

   Timer_Resource_Error : exception;

private
   ... -- not specified by the language
end Ada.Execution_Time.Timers;
```

Listing D.6: *Group budget definition*

```
with System;
with System.Multiprocessors;
package Ada.Execution_Time.Group_Budgets is

   type Group_Budget (CPU : System.Multiprocessors.CPU :=
                                    System.Multiprocessors.CPU'First)
      is tagged limited private;

   type Group_Budget_Handler is access
         protected procedure (GB : in out Group_Budget);

   type Task_Array is array (Positive range <>) of
                                    Ada.Task_Identification.Task_Id;

   Min_Handler_Ceiling : constant System.Any_Priority :=
      implementation-defined;

   procedure Add_Task (GB : in out Group_Budget;
                       T  : in Ada.Task_Identification.Task_Id);
   procedure Remove_Task (GB: in out Group_Budget;
                          T  : in Ada.Task_Identification.Task_Id);
   function Is_Member (GB : Group_Budget;
                       T : Ada.Task_Identification.Task_Id) return Boolean;
   function Is_A_Group_Member
      (T : Ada.Task_Identification.Task_Id) return Boolean;
   function Members (GB : Group_Budget) return Task_Array;

   procedure Replenish (GB : in out Group_Budget; To : in Time_Span);
   procedure Add (GB : in out Group_Budget; Interval : in Time_Span);
   function Budget_Has_Expired (GB : Group_Budget) return Boolean;
   function Budget_Remaining (GB : Group_Budget) return Time_Span;

   procedure Set_Handler (GB     : in out Group_Budget;
                          Handler : in Group_Budget_Handler);
   function Current_Handler (GB : Group_Budget)
      return Group_Budget_Handler;
   procedure Cancel_Handler (GB      : in out Group_Budget;
                             Cancelled : out Boolean);

   Group_Budget_Error : exception;

private
      -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

Listing D.7: *Clock and alarm definition*

```
type Clock_Descriptor is
   record
      Base_Time : Time;
      -- Base time of clock

      First_Alarm : Alarm_Id;
      -- Points to the first alarm of this clock

      Capacity : Natural;
      -- Remaining alarm capacity, no more alarms if zero

   end record;

type Alarm_Descriptor is
   record
      Timeout : Time;
      -- Timeout of alarm when set

      Clock : Clock_Id;
      -- Clock of this alarm

      Handler : Alarm_Handler;
      -- Handler to be called when the alarm expires

      Data : System.Address;
      -- Argument to be given when calling handler

      Next : Alarm_Id;
      -- Next alarm in queue when set, null otherwise

   end record;
```