Reducing the Cost of Real-Time Software through a Cyclic Task Abstraction for Ada

Assisted by Acton: the novel Ada real-time executive

Patrick Bernardi

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University
February 2017

Declaration

I hereby declare this thesis is my own original work carried out exclusively for the degree of Doctor of Philosophy at the Australian National University. Contribution by others though feedback, editing and prior work are acknowledged in the customary manner.

Patrick Bernardi
February 9, 2017

© Patrick Bernardi 2017

Abstract

Ada is unique as a software engineering language, facilitating the development of reliable and maintainable software through clear, unambiguous, modular code enforced to a specification. However, despite comprehensive real-time systems support, Ada lacks the abstraction at the heart of these systems: cyclic tasks. Without it, users resort to solutions decomposing their designs: introducing unnecessary complexity to a real-time systems primitive. Furthermore, the only Ada executive for microcontrollers, gnat for Bare Boards, impedes the timing analysis of tasks and schedulability analysis of systems. Consequently, Ada and its environments do not reach their potential of producing low-cost, reliable and maintainable real-time systems.

This thesis unlocks this latent potential through the development of the *Cyclic Task Specification* and Acton. The *Specification* extends Ada to incorporate cyclic tasks within Ada's existing task abstraction using a simple but powerful model. Acton, on the other hand, is a new Ada executive built from the ground up to support real-time systems, forgoing an existing Ada tasking run-time in favour of building the tasking semantics natively within its kernel. The result is a flexible and portable Ada executive which correctly apportions execution time and enforces cyclic task attributes. Furthermore, tasks only perform user assigned activities and lower priority tasks cannot interrupt a running task.

Their contributions come at the cost of a more complex language and larger execution time overheads for kernel operations. For Acton, the up to order of magnitude larger overheads is the cost of simplifying task timing and system schedulability analysis. For the *Cyclic Task Specification*, the added complexity simplifies the expression of cyclic task patterns for a wide range of systems. Its simple model centred on task attributes permits model expansion outside the formal language, addressing aversions to incorporating high-level abstractions.

Together, the *Cyclic Task Specification* and Acton advance real-time systems by providing the clarity, structure and simplicity needed to express cyclic tasks and have their real-time constraints enforced at run-time. Furthermore, real-time analysis tools can now use the same cyclic task semantic information as the compiler to provide automatic timing verification and run-time enforceable cyclic constraints. Consequently, the contributions of this thesis allow real-time systems to use less code and have less scope for faults: reducing the life-cycle cost of real-time systems through quicker development and maintenance.

Acknowledgements

I WOULD NOT HAVE EMBARKED DOWN THE LONG JOURNEY OF A POSTGRADUATE degree if not for my supervisor Uwe Zimmer. His convincing arguments for a topic on autonomous model helicopters proved irresistible compared to my then favoured option of working in telecommunication industry. For the reader this will be puzzling because, as presented, this thesis has nothing to do with autonomous model helicopters. However, when push came to shove on a failing autonomous model helicopter project, I switched to a question that was gnawing at me more than the failed autonomous electronic hardware I had in my hands:

> *Why was there no Ada tasking run-time for my rather powerful microcontroller and why does Ada not support the high-level real-time abstractions I was taught and used in my designs?*

The origin of this question lies very much with Uwe, but from his role as a teacher rather than as a supervisor. Uwe taught Ada and real-time systems in two courses I took as an undergraduate. His style of teaching led me to appreciate and value the power Ada and real-time systems bring to the building of reliable, maintainable and predicable systems. However, the hallmark of a great teacher is not the knowledge imparted but the inspiration and curiosity planted to seize on the gained knowledge. Uwe achieved this, instilling the expectation of having the right tools for the task at hand. Uwe is a teacher par excellence, whom I thank for his teaching, and his ongoing advice, appraisal and support throughout the thesis.

I am also grateful to acknowledge another teacher par excellence from my youth, Frank O'Shea, for his influence and passion for mathematics.

This thesis, however, needed more than determination, inspiration and support to make it possible: it needed a high quality Ada environment from which my ideas could integrate. AdaCore delivered this in the form of the GNAT Ada environment and I thank the many people who developed it for producing such a high-quality and accessible open-source Ada environment. It is a testament to its design that someone without any experience developing compilers can step-in and learn not just how the GNAT compiler works, but also how compilers work generally. Furthermore, exposure to GNAT has taught me how to properly write software, particularly for large systems, for which I am thankful.

Nevertheless, I cannot diminish the immense support by others. To my fellow PhD students Florian Poppa and Ben Coughlan I thank for the amazing office atmosphere and friendships. Our office was one of engaging discussions, encouragement and support: providing a valuable environment for sounding board ideas.

To the participants of IRTAW-17 I thank for the warm welcome I received to the small community, and for their invaluable input to refine the rough edges of the *Cyclic Task Specification*.

I would like to end by thanking my friends and family who provided vast support and encouragement over the course of my thesis; each in their own cherished way. In particular, special mention goes to Paul Altin and Steve Peterson. To my brother James and sister Maria I thank them for reading sections of my thesis and making my life more fulfilling.

Finally, to my parents Linda and John, I thank them for their unwavering support and generosity. Without them, I would not be where I am today.

Contents

Declaration i

Abstract iii

Acknowledgements v

Contents vii

Chapter 1 Introduction 1
- *Task Abstraction* 2
- *Task Abstraction Implementation* 3
- *Outline* 5
- *Thesis Conventions* 6

Chapter 2 Cyclic Tasks and Ada 9
- *Cyclic Task Definitions and Properties* 11
- *Ada's Real-Time Facilities* 15
- *Implementing Cyclic Tasks in Ada* 16
- *Prior Work Incorporating Cyclic Tasks into Ada* 30
- *Cyclic Tasks in Other Languages* 32
- *Conclusion* 38

Chapter 3 The Cyclic Task Specification 41
- *The Specification* 42
- *Examples* 51
- *Implementation* 54
- *Ada Semantic Interface Specification* 56
- *Language Impact and Flexibility* 62
- *Comparison with Existing Ada Approaches* 65
- *Conclusion* 72

Chapter 4 Review of gnat for Bare Boards for Real-Time Systems 75
- *gnat* 76
- *gnat for Bare Boards* 79
- *Real-Time Problems in gnat for Bare Boards* 84
- *Conclusion* 89

Chapter 5	**Acton**	**91**
	Rationale	*92*
	Architecture	*93*
	Oak Resources	*96*
	The Oak Run-Loop	*112*
	Scheduling	*116*
	Task Management	*123*
	Cyclic Tasks	*128*
	Protected Objects	*134*
	Interrupt Handling	*143*
	Conclusion	*148*
Chapter 6	**Review of Acton**	**151**
	Platforms and Setup	*152*
	Kernel Design	*154*
	Scheduling and Task Dispatching	*158*
	Idle State	*163*
	Protected Objects	*164*
	Interrupt Handling	*165*
	Memory Footprint	*166*
	Portability	*169*
	Language Cover	*169*
	Conclusion	*172*
Chapter 7	**Conclusion**	**175**
	Future Work	*179*
Appendix A	**Cyclic Task Libraries**	**181**
Appendix B	**References**	**189**

1
GAUI X5
3D master club

Chapter 1

Introduction

A gap exists between the design and specification of tasks in a real-time system and their implementation in Ada. Predominately, these tasks are inherently cyclic in nature, yet Ada does not provide a simple way to express cyclic tasks and their properties. Furthermore, microcontroller-based real-time systems lack a suitable Ada executive or run-time with the capability to enforce cyclic task properties online, nor fully support the needs of real-time users.

These observations resulted from experiences gained at the beginning of the doctorate, when the dissertation began as a study into the reliability of autonomous scale helicopters. At the time a suitable autonomous platform was not readily or affordably available, necessitating its development for the thesis.

While abandoned because of hardware issues, the software of the autonomous helicopter project inflicted a desire to address the observed weaknesses of cyclic tasks in Ada. Ada was chosen because, unlike other languages, Ada marks itself as a language aiding software engineering practises: facilitating the development of reliable and maintainable software by promoting the writing of clear, unambiguous and modular code, enforced to a specification (Moralee, 1981). Consequently, Ada minimises the life-cycle cost of software (Whitaker, 1993) and naturally befits a project where the software must be correct or people get hurt: a scale helicopter has all the inherent safety of an upside-down flying lawnmower.

An additional appeal of Ada lay with its unique task abstraction: a syntactical and semantic approach placing the state of a task and its operations into a single type. This approach marries with the architecture of real-time systems where systems decompose into a set of independent tasks — each with their own individual timing constraints. However, a mismatch exists between the task abstraction Ada provides and the task abstraction real-time systems use.

Furthermore, Ada's tasking support comes in the form of an Ada run-time on top of an existing operating system or as an Ada executive implementing Ada tasks natively within its kernel. For the autonomous helicopter project, there was not a readily available Ada executive for the hardware's chosen microcontroller: a Power Architecture-based Freescale MPC5554. An open source Ada executive based on GNAT and the Ravenscar Profile does exist, presenting an opportunity to bring the executive to the MPC5554. However, its design complicates the calculation of a task's worst-case execution time (WCET) and the schedulability analysis of the system. Moreover, the constraints imposed by the Ravenscar Profile prevent online WCET enforcement.

In response, this dissertation has developed a cyclic task abstraction for Ada and a new Ada real-time executive. The *Cyclic Task Specification* defines the new cyclic task abstraction for Ada: incorporated within the language's existing syntax and semantics to permit a seamless transition from the design of a real-time system to its implementation and back again. Acton is the dissertation's new Ada real-time executive: integrating Ada's dynamic tasking semantics into a novel kernel. The new executive simplifies the timing analysis of tasks and the schedulability analysis of real-time systems, while providing online enforcement of cyclic task attributes. Together, the new abstraction and executive aims to improve the reliability, maintainability and productivity of real-time systems: aiding Ada's ultimate goal of reducing the life-cycle cost of software.

Task Abstraction

During system design, each real-time task is specified with a set of timing, release and constraint properties. To implement them, Ada provides a rich set of real-time facilities that allows users to build sophisticated cyclic tasks to these specifications. Ada can build cyclic tasks with different release mechanisms; build tasks that detect and respond to missed deadlines and over-running their execution bounds; and safely mix aperiodic tasks with periodic and sporadic tasks through the sharing of CPU budgets. The problem Ada has in realising these cyclic tasks lies with the word *build*: the real-time facilities provided by Ada, while powerful, are low-level system primitives forming only the building blocks users use to create cyclic tasks themselves — a process repeated for each cyclic task. This approach is not desired as it is error prone, a poor use of time, and makes software harder to read and maintain. In essence, the current Ada approach to cyclic tasks is akin to languages that do not support concurrent features within their syntax and instead rely on the assembly of library calls: an approach lacking the simplicity, clarity, and semantic protections offered by an integrated approach.

Thus, a gap exists between the high-level cyclic task abstraction real-time systems are designed around and the lower-level task abstractions provided by

Ada: imposing extra effort and risk on the development of real-time systems by forcing the decomposition of the real-time model into an Ada model.

hypothesis

From observations made in Chapter 2, this dissertation forms the hypothesis that incorporating a cyclic task abstraction into Ada's syntax and semantics increases the reliability and maintainability of real-time systems; ultimately serving the central purpose of Ada: minimising the life-cycle cost of software.

To support the hypothesis, this dissertation has created an extension to Ada called the *Cyclic Task Specification* that integrates a cyclic task abstraction into Ada. The abstraction consists of an extension to Ada's existing task syntax and a cyclic task model providing the abstraction's semantics. The *Specification* will be shown to increase the reliability and maintainability of real-time Ada software since the cyclic task design abstraction becomes directly implementable in Ada, removing the existing need to decompose cyclic tasks into low-level primitives during development and their reconstruction during program maintenance. Consequently, a program consists of less code: reducing the effort to read and write real-time programs.

Task Abstraction Implementation

The other side of Ada's tasking model is its implementation, requiring run-time support to underpin the language's dynamic tasking semantics. For this dissertation's first hypothesis, a tasking implementation incorporating the *Cyclic Task Specification* in the first instance validates the feasibility of the language extension. Secondly, an implementation allows real-time systems to immediately take advantage of the benefits the *Cyclic Task Specification* provides. Thus, this dissertation proceeds with an implementation of the *Specification*, with a focus on microcontrollers because of their common use in real-time systems (Laplante & Ovaska, 2011) and due to the background of the dissertation.

Implementation of the *Cyclic Task Specification* occurs inside GNAT, an Ada environment developed by AdaCore. GNAT was chosen because it is the only open source Ada environment and the only environment supporting the current Ada standard ISO/IEC 8652:2012. Commonly known as Ada 2012, the *Cyclic Task Specification* leverages the current standard to provide a natural expression of cyclic tasks within Ada.

Extending GNAT with *Cyclic Task Specification* required modification to GNAT's compiler and run-time. Its integration into the compiler was straightforward due to the excellent design and clarity of the GNAT compiler. Therefore, the dissertation does not detail the changes made to the compiler to directly support *Cyclic Task Specification*, with *GNAT: The GNU Ada Compiler* (Miranda & Schonberg, 2004) and the compiler sources providing the necessary resources explaining how to extend the compiler.

By contrast, extending GNAT's run-time was not straightforward. Implementing the dynamic semantics of Ada, the GNAT Run-Time Library depends heavily on the underlying operating system to provide the required tasking services (AdaCore, 2015c; Giering & Baker, 1994). With the Freescale MPC5554 selected

as the target microcontroller, two options existed to supply the needed tasking services:

- RTEMS, a small open source real-time operating system (RTOS) supporting the GNAT run-time (OAR, 2011).
- A bare boards version of the GNAT run-time that replaces the run-time's operating system dependencies with a purpose built kernel, turning GNAT into an Ada executive (AdaCore, 2015B). Developed by AdaCore, this Ada executive appears to lack an official name, instead marketed as *GNAT for [Processor]* or *GNAT for Bare Board [Processor]* — for example *GNAT for LEON* or *GNAT for Bare Board ARM*. For the purpose of this dissertation, this variant of GNAT is called *GNAT for Bare Boards* (GNAT-BB).

Being a general-purpose RTOS not devoted to the needs of an Ada run-time means RTEMS imposes larger overheads and greater timing complexity than GNAT with a specialised run-time (de la Puente, Ruiz & González-Barahona, 1999). For this reason this dissertation did not consider GNAT on RTEMS. MaRTE, another RTOS built to support Ada applications (Rivas & Harbour, 2001), was not considered since the RTOS only supports X86 platforms and carries similar overheads as RTEMS by offering POSIX.13 services and an extensive set of drivers. This differs from GNAT-BB low-overhead approach of directly implementing only the services required by GNAT.

Before incorporating *Cyclic Task Specification* support into GNAT-BB, this dissertation reviewed the suitability of GNAT-BB as a real-time executive supporting cyclic tasks. Presented in Chapter 4, the review found limitations in GNAT-BB and the GNAT run-time that unreasonably complicated the timing of tasks and the schedulability analysis of systems. Additionally, GNAT-BB compromises run-time enforcement of a task's WCET with the task's execution time including actions not related to the task.

These limitations arise from the design of GNAT-BB's kernel and run-time. In particular, design decisions made to accommodate GNAT's run-time portability creates negative consequences for real-time systems. In addition, GNAT-BB adherence to the Ravenscar Profile meant the Ada executive did not offer a mechanism to react to tasks exceeding their execution time, nor offered any support for different task dispatching policies or execution servers without significant changes to the GNAT-BB kernel.

hypothesis

From the review of GNAT-BB in Chapter 4, this dissertation hypothesises a new Ada executive built specifically for real-time systems will simplify system timing and schedulability analysis, and enable run-time enforcement of cyclic task attributes with tighter WCET bounds.

To support the second hypothesis, the dissertation has developed a new Ada real-time executive called Acton. Acton focuses on the needs of real-time users with its design simplifying the timing analysis of tasks and schedulability analysis of systems. Acton differs from GNAT-BB by integrating Ada tasking support within its kernel — creating a new Ada tasking run-time. The Ada executive supports the majority of the *Cyclic Task Specification* and goes outside the Ravenscar Profile to support multiple task dispatching policies and to relax many protected object restrictions.

Outline

Support of the dissertation's first hypothesis begins with an examination of cyclic tasks in Chapter 2. After defining the cyclic task model and describing the importance of cyclic tasks for real-time systems, the chapter demonstrates the implementation of cyclic tasks in Ada today and the subsequent techniques developed to reduce their implementation complexity. In doing so, Chapter 2 provides evidence of the gap between real-time theory and Ada practise, and the negative implications the gap has on the reliability and maintainability of real-time systems. The Chapter will conclude with a review of earlier proposals to incorporate cyclic tasks into the semantics of Ada and will review the approach other languages have taken to incorporate cyclic tasks.

Chapter 3 removes this gap through the *Cyclic Task Specification*: extending Ada's syntax and semantics to include a cyclic task abstraction. This chapter will demonstrate how extending Ada's existing task abstraction creates a superior solution to implementing cyclic tasks over existing library-based approaches. Importantly, the *Specification* works with the existing language to minimise impact on the language and its existing users, offering cyclic tasks in a manner familiar to Ada users.

Focus of the dissertation switches to the rationale for the second hypothesis in Chapter 4, reviewing GNAT-BB's suitability for real-time systems as a precursor to porting it to the MPC5554 and incorporating the *Cyclic Task Specification*. This chapter centres on GNAT-BB's scheduling, protected object, and execution time support and how GNAT-BB's design in these areas affects real-time systems.

Chapter 5 presents *Acton*, the thesis' new Ada real-time executive designed to explicitly support real-time systems. This chapter documents Acton's design and implementation, focusing on the executive's novel aspects and how it supports both real-time systems and the *Cyclic Task Specification*.

With an Ada executive designed to support real-time system in hand, Chapter 6 reviews how Acton serves the needs of real-time users compared to GNAT-BB. The review also reflects on the merits of both Ada executives as general bare-board Ada run-times.

The thesis concludes with Chapter 7: evaluating the presented hypothesises in light of the *Cyclic Task Specification* and Acton. Also discussed is the future direction for both the *Specification* and Acton.

Thesis Conventions

Ada as standardised as ISO/IEC 8652:2012 (Information technology — Programming languages — Ada) is referred to in this dissertation as Ada or, if needed to distinguished it from earlier editions, Ada 2012. Previous editions and revisions of Ada are denoted by their year of standardisation. The standard itself is referred to as the Ada Reference Manual or ARM. References to the ARM are done by chapter and section, for example ARM 9.1 for *Task Units and Task Objects*.

Figure 1 describes how to interpret the control flow diagrams used by this dissertation.

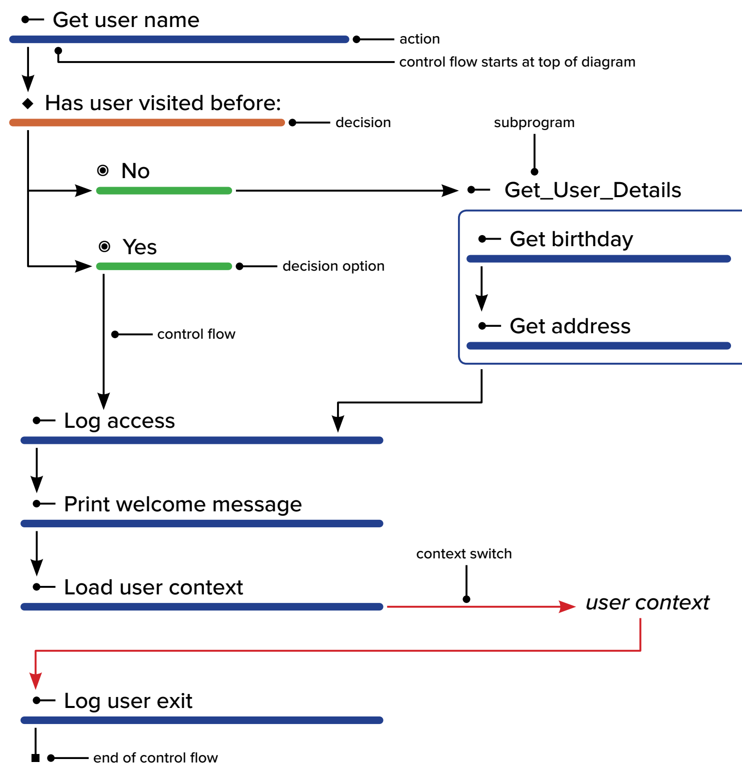


Figure 1
A control flow diagram for an example system.

2
NXT
mindstorms

Chapter 2

Cyclic Tasks and Ada

Like all embedded systems, real-time systems have computer hardware integrated into their control loops. They utilise this hardware to calculate and drive appropriate actuator responses in reaction to the environment state gathered by the system's sensors: thus permitting a system to react and modify its environment to meet its design requirements (Laplante & Ovaska, 2011).

Real-time systems distinguish themselves from other embedded systems with the timeliness of the system's results as important as their logical correctness (Stankovic, 1988). Hard real-time systems consider violation of a system's timing constraints erroneous, while soft real-time systems can cope with some violation with degraded service; potentially placing less emphasise on the results missing their deadlines (Buttazzo, 2011; Manacher, 1967). Firm real-time system form an extreme soft-real time system subset with results discarded if they miss their deadline.

Take, for example, a humanoid lego Mindstorms robot named Frank: a simple but non-trivial real-time system used by the dissertation to develop and demonstrate both the *Cyclic Task Specification* and Acton. Frank has a simple existence, walking in (vaguely) straight lines and turning only to avoid hitting obstacles. Equipped with an ultrasonic distance sensor to gauge the distance to an obstacle in front of it, Frank is a hard real-time system because failure to react to the presence of an obstacle in time will cause the robot to hit it.

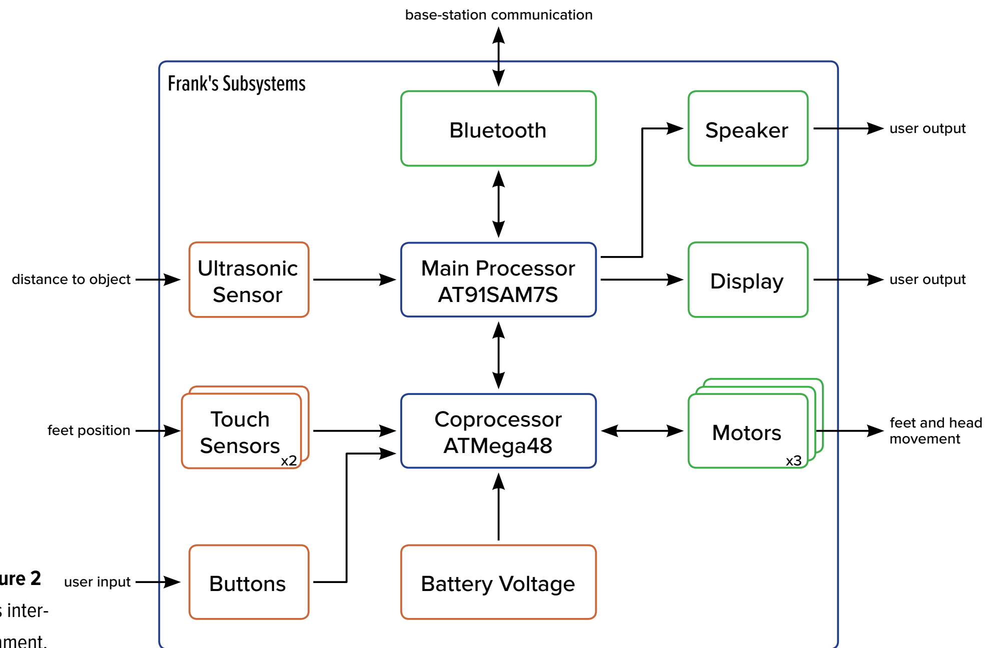


Figure 2 Frank's subsystems and its interaction with its environment.

While simple, Frank consists of numerous subsystems collecting data and modifying the state of robot and the environment it operates in (Figure 2) — including user interaction through the robot's buttons, screen and speaker. Consequently, the software running on the main processor consists of the required control loop calculations and the interactions with the various subsystems (Kopetz, 2011). The functionality of the system decomposes into a set of tasks the robots software will perform, with each task focusing on a subsystem interaction or control calculation (Cottet, Delacroix, Kaiser & Mammeri, 2002). Each task possesses individual real-time constraints, with a hard real-time system able to consist of a collection of hard, soft and non-real-time tasks. For example, Frank has a hard real-time task to collect obstacle distance measurements while the task performing screen updates has soft real-time requirements.

The key feature of such a real-time system is the performance of the same sequence of tasks in response to changes to the system's inputs: resulting from the integration of embedded software into the system's control loop. Thus, the tasks themselves are repetitive, commencing a new cycle each time the digital control loop they participates in runs. Because of their repetitive nature, this dissertation calls these tasks *cyclic tasks* to differentiate them from *sequential tasks*; that is, a task that executes once. This dissertation makes a clear distinction between the two kinds of tasks because Ada's existing task type only implements a sequential task abstraction.

Because real-time tasks are cyclic and their timing behaviour known offline, it becomes possible to determine if all tasks in a system will meet their timing constraints before running the system (C. L. Liu & Layland, 1973). This is the most important aspect of cyclic tasks, as a schedulability analysis can provide a guarantee a system is free from timing-related faults; improving system reliability and reducing development costs as problems are found and dealt with during the initial development (Sha et al., 2004).

This chapter reviews the support for cyclic tasks in Ada, beginning by defining key cyclic task definitions and properties as used by the dissertation. The

chapter then introduces Ada's real-time facilities and how users assemble them to create cyclic tasks. The chapter will then demonstrate how this assembly is detrimental to the production of reliable and maintainable real-time systems, and how existing efforts to reduce the complexity through library-based approaches similarly fall short. Finally, the chapter will review previous attempts to integrate cyclic tasks into Ada and other languages.

Cyclic Task Definitions and Properties

Classical real-time theory defines *tasks* in the general sense of the word: as a unit representing an activity or algorithm (Cottet et al., 2002; Kopetz, 2011). A *job* represents the single execution of a task, and thus under the classical real-time model a task consists of an infinite number identical jobs with the system releasing a job in response to a system event (Buttazzo, 2011; Laplante & Ovaska, 2011). Each job is characterised by its *arrival time* to the scheduler, its *computation time* and the *deadline* the job has to complete by.

The unit of work under the classical real-time model is the job. Without any restrictions on the release of jobs, a task may have more than one job executing at the same time. Allowing jobs to overlap requires they execute inside their own context, imposing significant complexity on real-time systems. In particular, there are data integrity issues with multiple jobs accessing the state of a task in addition to resource contention between jobs of the same task.

These problems disappear by imposing the restriction that the arrival time of a job can be no earlier than the deadline of the preceding job. Consequently, the restriction enables tasks to become the work unit since only one job can execute at any point in time.

Such a restriction enables the dissertation to define a more precise definition of a *task*: an execution unit encapsulating the code and state of an activity or algorithm. The new definition forms the basis of the dissertation's cyclic task model and aligns with same definition used by Ada (Ichbiah, Barnes, Firth & Woodger, 1986).

The dissertation's cyclic-time model deviates from the classical real-time model because of the refined task definition. Aside from the imposed deadline restrictions, the primary difference lies in the terminology used to describe real-time tasks and their attributes to align closely with the new definition.

Dissertation's Cyclic Task Model

definition

Cyclic task A task whose code executes in response to an event. Such a task sleeps while waiting for the next event after completing a *cycle* of its code. A cyclic task requires explicit termination to terminate the task. By contrast, a *sequential task* executes its code once and then terminates.

Cycle The execution of a cyclic task's code once. Similar to the concept of a job used by the classical real-time model except a task performs the *cycle* itself.

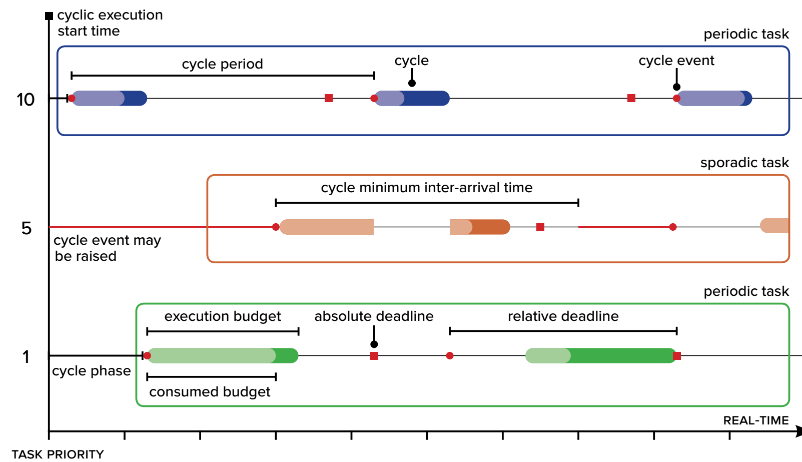


Figure 3
Graphical representation of cyclic task properties.

Real Time The physical time as observed in the environment the system is apart of (ARM D.8). It is non-decreasing monotonic.

Real-Time Clock A monotonic clock incremented at a steady rate from which time within a real-time system derives.

Cyclic Execution Start Time The earliest time in a system which all tasks may begin executing their first cycle.

Phase An offset from the cyclic execution start time before the first cycle may begin executing.

Worst-Case Execution Time (WCET) Upper bound on the *execution time* for a task's cycle. The execution time for a task is the time the task spends executing it code on a processor. Known under the classical real-time model as a job's *computation time.*

Execution Budget Total amount of execution time a cycle may consume. Normally set to the WCET of the task.

Relative Deadline The duration from a cycle event a task has to complete by.

Absolute Deadline The time a task has to complete the current cycle by, being the sum of the task's relative deadline and the time of the cycle event that started the cycle.

Cycle Event The event causing a cyclic task to execute a cycle. There are four types of cycle events:

- *Periodic events* repeat at fixed intervals given by the task's *cycle period* and sourced from the system's real-time clock. A cyclic task with periodic cycle events is called a *periodic task.*
- *Aperiodic events* repeat at irregular intervals and occur from the activities of other tasks or external interrupts. A cyclic task with aperiodic cycle events is called an *aperiodic task.*
- *Sporadic events* are a type of aperiodic event where a *minimum inter-arrival time* (MIT) exists between events. A cyclic task with sporadic cycle

events is called a *sporadic task*.

- *Cycle completion events* occurs when a cycle completes. A cyclic task commencing a new cycle in response to its own cycle completion event is called a *yielding task*: a task that yields the processor between cycles. The dissertation's real-time model introduces yielding tasks to extend the use of the cyclic task model outside the real-time domain, covering applications requiring tasks to execute cycles back-to-back or need a more complex event release mechanism than provided by other cyclic task types.

Priority A value assigned to each task denoting the precedence over other tasks to execute on a processor or processors.

real-time scheduling

A scheduler organises the execution of a given task set: determining when a given task executes on a processor in the system (Cottet et al., 2002). A real-time scheduler focuses on ensuring the tasks within a task set meet their real-time constraints, achieved by assigning each task a priority (Buttazzo, 2011). At a *task dispatching point*, where the system selects and runs a task, a real-time system dispatches the highest priority task eligible to run.

Schedulers fall into a number of different categories:

Task dispatching point location A *non-preemptive* scheduler limits task dispatching points to locations of the task's choosing. By contrast, a *preemptive* scheduler imposes its own task dispatching points; for example when another task wakes or when a task has used up its allotted processor time.

Task priority assignment A *fixed priority scheduler* has task priorities assigned offline as part of the system's development, with the goal to create a feasible schedule that permits all tasks to meet their real-time constraints. At run-time, the scheduler simply dispatches the highest priority task at each task dispatching point. A *dynamic priority scheduler* on the other hand assigns priorities at run-time, assigning priorities based on the urgency to complete the task set at the given task dispatching point.

Number of processors The complexity involved with dispatching to multiple processors means different scheduler algorithms are better suited to *uniprocessor*, *partitioned multiprocessor* or *global multiprocessor* systems.

Real-time schedulers provide the ability for users to determine the *feasibility* of a set of tasks to meet their timing constraints under a scheduler by using a *schedulability analysis* (Sha et al., 2004). The ability to perform a schedulability analysis is essential for hard real-time systems as the analysis determines if the system is free of timing faults and relies on the cyclic nature of real-time systems, in particular the periodicity of its tasks. If all the tasks are periodic, then a minimum set of cycles across all tasks exist until the set repeats. Thus, if a schedule is feasible for the minimum set, it is feasible for any set.

Sporadic tasks fit within a schedulability analysis by treating them like periodic tasks, using knowledge of their MIT and event frequency (Audsley, Burns, Richardson, & Wellings, 1991). By contrast, aperiodic tasks by definition have no periodicity attached to them and require *execution-time servers* (or *execution*

servers for short) to provide periodic access to the processor (Burns & Wellings, 2006). An execution server operates like a periodic task — with an attached periodicity, deadline and execution budget — except the server does not dispatch to the processor: instead, a task using the server is. Execution servers are also referred to in literature as *priority servers* (Buttazzo, 2011) or *aperiodic servers* (Lehoczky & Ramos-Thuel, 1992).

As a mature research field, the real-time scheduler domain comes with extensive research and textbooks documenting different real-time schedulers and execution servers. Books from Burns and Wellings (2009), Buttazzo (2011), Cottet et al. (2002) and Liu (2000) provide detail descriptions of common real-time schedulers, while Sha et al. (2004) and Davis and Burns (2011) provide comprehensive literature reviews on uni and multiprocessor schedulers.

run-time deadline and execution budget enforcement

Task deadlines and execution budgets lie at the heart of real-time systems to ensure tasks meet their timing constraints. While a schedulability analysis can provide a guarantee a system has no timing faults for a given set of tasks, the guarantee only holds while the assumptions underpinning the analysis is correct.

For hard real-time systems theses assumptions include: no unexpected transient or permanent hardware faults causing a task to run longer than expected; no logic defects in the software that may cause it or another task to run longer than its bounds; and no external communication exceeds its time bounds. On the other hand, soft real-time systems explicitly assume missed deadlines will occur. Consequently, soft real-time systems need to detect missed deadlines to know when to devalue results, while hard real-time systems require deadline enforcement to handle situations where the system assumptions fail.

Without WCET enforcement, T2 will be blamed for the system error when the problem lies in T1.

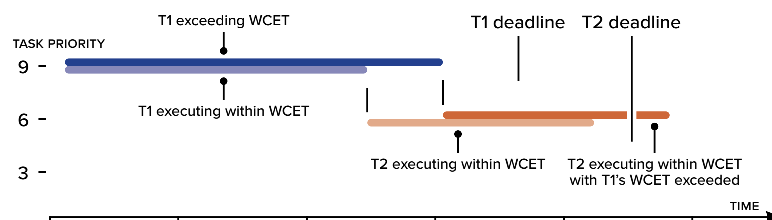


Figure 4
Not enforcing T1's WCET causes T2 to miss its deadline. Reacting to T2 also masks the fault in T1.

Additionally, real-time tasks require execution budget enforcement: if a task exceeds its execution budget due to a temporary or permanent hardware fault, the system looses its timing guarantees and a task may miss its deadline (Kopetz, 2011). Catching execution budget exhaustion before a missed deadline allows a system to identify and treat the task with the timing anomaly, as opposed to potentially having to resolve its effect on another task (Figure 4). It also help to easily identify the causes of timing violations, with permanent or transient hardware faults more likely to cause a task to exhaust its execution budgets before it misses a deadline, while a missed deadline would more likely occur from a task waiting longer on the activity of another task or external subsystem.

Ada's Real-Time Facilities

Ada 2012 provides thorough support for real-time systems, with the base language assisting the prime software engineering requirements for real-time systems: reducing the life-cycle cost of software by facilitating reliable and maintainable software. Ada then provides explicit real-time support through an optional *Real-Time Systems Annex* (ARM Annex D).

Ada's tasks form the backbone of the language's real-time support, using the same basic task definition as the dissertation: encapsulating the state of the task and the code that operates on it within a single programming unit (Ichbiah et al., 1986; ARM 9.1). A public specification and private body provide this abstraction using the same approach as used by packages and protected objects. Software benefits from this abstraction as it provides a new scope from which declarations relevant only to the task can be made, resulting in clearer and easier to maintain programs. Additionally, the scope ensures a task cannot access another task's internal state.

To provide inter-task communication and shared resource access, the core language provides protected objects: an abstraction coordinating access to shared data (ARM 9.4). Additionally, the core language provides a flexible delay statement allowing a task to sleep for a specific period or, more usefully for real-time applications, until a particular point in time passes (ARM 9.6).

The *Real-Time Systems Annex* extends Ada's basic tasking model to include a comprehensive and flexible priority, task dispatching and monotonic time models in addition to incorporating a substantial library of real-time abstractions. The *Annex* priority model forms the backbone of Ada's real-time support, clearly defining how the language uses priorities and detailing the situations when a priority may inherit from another source (ARM D.1).

Complementing the priority model, the *Real-Time Systems Annex* provides a task dispatching model based on conceptual priority-ordered ready queues (ARM D.2.1). The model defines how tasks from these ready queues execute and is sufficiently generic to permit the use of different schedulers under a single language defined approach. Responsibility for the these queues, and determining the time tasks are added or removed, lies with the task dispatching policies. These policies implement the scheduler algorithms, with the *Annex* providing four language-defined policies covering the most commonly used schedulers:

- **FIFO_Within_Priorities** A fixed priority preemptive scheduler
- **Non_Preemptive_FIFO_Within_Priorities** A fixed priority non-preemptive scheduler
- **Round_Robin_Within_Priorities** A time sliced preemptive scheduler
- **EDF_Across_Priorities** A dynamic priority preemptive scheduler

Finally, the *Annex* provides comprehensive additions to Ada's run-time library:

- Timing events (ARM D.15)

- Execution time (ARM D.14)
- Execution time timers (ARM D.14.1)
- Group execution time budgets (ARM D.14.2)
- Synchronous task control (ARM D.10)
- Synchronous barriers (ARM D.10.1)
- Processor assignment (D.16)
- Scheduling partitioning (D.16.1)

Since much of the rich and extensive tasking model Ada provides lies outside the needs of real-time users, the *Real-Time Systems Annex* also makes available optional tasking model restrictions a run-time system may use to facilitate the construction of efficient real-time systems (ARM D.7). The Ravenscar Profile (ARM D.13) collates a set of these restrictions under a single usage profile to enable the production of Ada programs with predictable timing behaviour, bounded memory usage and amenable to schedulability analysis (Burns, Dobbing & Vardanega, 2004).

Implementing Cyclic Tasks in Ada

While Ada shares a basic tasking model in line with this dissertation, the language only provides a sequential task abstraction. Ada instead expects users to build cyclic tasks upon these sequential tasks using its control statements, protected objects and *Real-Time Systems Annex* facilities. This section explores how users build cyclic tasks using this approach and documents the accompanying issues before examining library-based approaches attempting to abstract away the implementation complexity.

Direct Ada Implementation

A sequential task that iterates over same code and never terminates is easily created:

Listing 1
A sequential task that repeats its statements indefinitely.

```
task type Loopy_Task
   with Priority => Some_Priority;

task body Loopy_Task is
   ... task state ...
begin
   ... initialise task ...
   loop
      ... do something ...
   end loop;
end Loopy_Task;
```

The task can turn into a cyclic task by including a *wait for event statement*:

```
task type Cyclic_Task
   with Priority => Some_Priority;

task body Cyclic_Task is
   ... task state ...
begin
   ... initialise task ...
   loop
      [wait for event statement]
      ... cyclic code ...
   end loop;
end Cyclic_Task;
```

Listing 2
A general Ada implementation of a cyclic task.

The *wait for event statement* uses an Ada statement or library procedure to cause the sequential task to wait at the start of the loop until a cyclic event occurs. Potential *wait for event statements* include the **delay until** statement and entry calls.

A user can build all kinds of cyclic tasks from this template. For example, using the delay until statement as the *wait for event statement* creates a periodic task:

```
task type Periodic_Task
   with Priority => Some_Priority;

task body Periodic_Task is
   Period  : constant Time_Span := Milliseconds (500);
   Phase   : constant Time_Span := Milliseconds (100);
   Wake_Up : Time               := Clock + Phase;

   ... task state ...

begin
   ... initialise task ...
   loop
      delay until Wake_Up;
      Wake_Up := Wake_Up + Period;

      ... cyclic code ...
   end loop;
end Periodic_Task;
```

Listing 3
A simple periodic task using Ada primitives.

An aperiodic task can use a protected object or a suspension object to signal an event occurrence. The *wait for event statement* becomes the entry or procedure call holding the task until the object releases it. Suspension objects create simpler aperiodic tasks since the object does not require the creation of a new type:

```
Aperiodic_Event : Ada.Synchronous_Task_Control.Suspension_Object;

task Aperiodic_Task
   with Priority => Some_Priority;

task body Aperiodic_Task is
   ... task state ...
```

Listing 4
A simple aperiodic task using Ada primitives.

Listing 4 continued
A simple aperiodic task using Ada primitives.

```
begin
   ... initialise task ...
   loop
      Suspend_Until_True (Aperiodic_Event);
      ... cyclic code ...
   end loop;
end Aperiodic_Task;
```

Combining both the periodic and aperiodic task templates results in a simple sporadic task:

Listing 5
A simple sporadic task using Ada primitives.

```
Sporadic_Event : Ada.Synchronous_Task_Control.Suspension_Object;

task Sporadic_Task
   with Priority => Some_Priority;

task body Sporadic_Task is
   MIT     : constant Time_Span := Milliseconds (500);
   Phase   : constant Time_Span := Milliseconds (100);
   Wake_Up : Time               := Clock + Phase;

   ... task state ...

begin
   ... initialise task ...
   Set_False (Sporadic_Event)
   loop
      Suspend_Until_True (Sporadic_Event);
      delay until Wake_Up;
      Wake_Up := Wake_Up + MIT;
      ... cyclic code ...
   end loop;
end Sporadic_Task;
```

The problem with building cyclic tasks on top of lower level task abstractions lies with the negative impact the approach has on the life-cycle cost of real-time software. As the chapter has described, the tasking model real-time systems use is the cyclic task model. Thus under Ada, there is a direct development cost as developers have to spend time, and thus money, to build up a new cyclic task for each task in their real-time system. Furthermore, ramifications to system reliability and maintenance also affect life-cycle costs:

- A user has to write more code, increasing the scope for the introduction of errors.
- Using cyclic task templates to reduce coding burden affects reliability since the compiler cannot verify the correct use of the template.
- Awkward to create multiple copies of the same task, in particular for aperiodic and sporadic tasks which rely on external objects to signal a cyclic event. Using the aperiodic task for example, each new task object requires its own suspension object, permitting errors like assigning the wrong suspension object to a task to occur.

- More code makes it harder to understand the task, affecting maintenance.
- Furthermore, the code and state associated with making the task cyclic mixes with the cyclic task's code and state, decreasing code readability.
- Requires users to carefully read a task's body to determine the type of cyclic event used, affecting maintenance speed and presenting potential misidentification.
- Requires coding convention to make it easier to identify cyclic task properties during maintenance, which is unenforceable by the compiler.
- Understanding the type of cyclic task and its attributes requires both the task specification and task body. This differs from Ada's general approach of having static task properties identifiable on the task specification, reducing the need to share the body of the task.

A critical problem with building cyclic tasks on top of a set of low-level real-time abstractions is the code complexity grows with the utilisation of the cyclic task model, compounding the above effects. For example, if an exception needs raising in response to a sporadic event occurring before the expiry of a task's MIT, the simple suspension object needs replacing with a more complex protected object, with the *wait for event statement* adjusting accordingly:

Listing 6
A sporadic task where an exception is raised in the releasing task if the task is released before its MIT.

```
protected Event_Object is
   procedure Signal_Event;
   procedure Initialise (Earliest_First_Event : Time);
   entry Wait_For_Sporadic_Event (Earliest_Next_Event : Time);
private
   Earliest_Event_Time : Time;
   Event_Occurred      : Boolean := False;
   Initialised         : Boolean := False;
end Event_Object;

protected body Event_Object is
   procedure Signal_Event is
   begin
      if Clock < Earliest_Event_Time then
         raise Program_Error;
      else
         Event_Occured := True;
      end if;
   end Signal_Event;

   entry Wait_For_Sporadic_Event (T : Time)
     when Event_Occured is
   begin
      if not Initialised then
         raise Program_Error;
      end if;
      Event_Occured := False;
   end Wait_For_Sporadic_Event;
```

Listing 6 continued
A sporadic task where an exception is raised in the releasing task if the task is released before its MIT.

```
  procedure Initialise (T : Time) is
   begin
      Earliest_Event_Time := T;
   end Update_Earliest_Event_Time;
end Event_Object;

task Sporadic_Task
   with Priority => Some_Priority;

task body Sporadic_Task is
   MIT     : constant Time_Span := Milliseconds (500);
   Phase   : constant Time_Span := Milliseconds (100);

   Earliest_Arrival_Time : Time := Clock + Phase;

   ... task state ...

begin
   ... initialise task ...
   Event_Object.Initialise (Earliest_Arrival_Time);

   loop
      Earliest_Arrival_Time := Earliest_Arrival_Time + MIT;
      Event_Object.Wait_For_Sporadic_Event
        (Earliest_Arrival_Time);
      ... cyclic code ...
   end loop;
end Sporadic_Task;
```

Further complexity exists when enforcing execution budgets and deadlines, requiring the incorporation of execution time timers and timing events into the cyclic task templates. For example, modifying the simple periodic task in Listing 3 to enforce budgets and deadlines results in:

Listing 7
A periodic task with enforced deadlines and execution budgets.

```
protected Timing_Enforcement is
   procedure Missed_Deadline (Event: in out Timing_Event);
   procedure CPU_Budget_Exceeded (TM : in out Timer);
end Timing_Enforcement;

protected body Timing_Enforcement is
   procedure Missed_Deadline (Event: in out Timing_Event) is
   begin
      ... Handle missed deadline ...
   end Missed_Deadline;

   procedure CPU_Budget_Exceeded (TM : in out Timer) is
   begin
      ... Handle exceeded execution budget ...
   end CPU_Budget_Exceeded;
end Timing_Enforcement;

type Deadline_Event is new Timing_Event with record
   T : Task_Id;
end record;
```

```
task type Periodic_Task with Priority => Some_Priority;

task body Periodic_Task is
   Period            : constant Time_Span := Milliseconds (500);
   Phase             : constant Time_Span := Milliseconds (50);
   Relative_Deadline : constant Deadline  := Milliseconds (400);
   CPU_Budget        : constant Time_Span := Milliseconds (100);

   Wake_Up          : Time := Clock + Phase;
   CPU_Timer        : Timer  (Current_Task'Access);
   Deadline_Timer   : Deadline_Event;
   P_Control        : Timing_Enforcement;
   Timer_Cancelled  : Boolean;

   ... task state ...
begin
   ... initialise task...

   Deadline_Time.T := Current_Task;
   loop
      Set_Handler (Event   => Deadline_Timer,
                   At_Time => Wake_Up + Realative_Deadline,
                   Handler => P_Control.Missed_Deadline'Access);
      delay until Wake_Up;
      Wake_Up := Wake_Up + Period;
      Set_Handler (TM      => CPU_Timer,
                   In_Time => CPU_Budget,
                   Handler =>
                           P_Control.CPU_Budget_Exceeded'Access);
      ... cyclic code ...
      Cancel_Handler (Event     => Deadline_Timer,
                      Cancelled => Timer_Cancelled);
      Cancel_Handler (TM        => CPU_Timer,
                      Cancelled => Timer_Cancelled);
   end loop;
end Periodic_Task;
```

Listing 7 continued
A periodic task with enforced deadlines and execution budgets.

Finally, different task dispatching policies may require different *wait for event statements*. The examples presented so far work under Ada's FIFO_Within_Priorities and Non_Preemptive_FIFO_Within_Priorities policies but not with the EDF_Across_Priorities policy. The latter policy requires the task to update the deadline stored with scheduler every iteration, accomplished via the Ada.Dispatching.EDF.Delay_Until_And_Set_Deadline procedure. Thus, the task body of the previous example becomes:

```
task body Periodic_Task is
   Period            : constant Time_Span := Milliseconds (500);
   Phase             : constant Time_Span := Milliseconds (50);
   Relative_Deadline : constant Deadline  := Milliseconds (400);
   CPU_Budget        : constant Time_Span := Milliseconds (100);

   Wake_Up          : Time := Clock + Phase;
   CPU_Timer        : Timer  (Current_Task'Access);
   Deadline_Timer   : Deadline_Event;
```

Listing 8
Listing 7's periodic task, modified to run under the EDF_Across_Priorities task dispatching policy.

Listing 8 continued
Listing 7's periodic task, modified to run under the EDF_Across_Priorities task dispatching policy.

```
   P_Control        : Task_Control;
   Timer_Cancelled  : Boolean;

   ... task state ...

begin
   ... initialise task...

   Deadline_Time.T := Current_Task;
   loop
      Set_Handler (Event   => Deadline_Timer,
                   At_Time => Wake_Up + Realative_Deadline,
                   Handler => P_Control.Missed_Deadline'Access);
      Delay_Until_And_Set_Deadline
        (Delay_Until_Time => Wake_Up,
         Deadline_Offset  => Relative_Deadline);
      Wake_Up := Wake_Up + Period;
      Set_Handler (TM      => CPU_Timer,
                   In_Time => CPU_Budget,
                   Handler =>
                           P_Control.CPU_Budget_Exceeded'Access);

      ... cyclic code ...

      Cancel_Handler (TM        => CPU_Timer,
                      Cancelled => Timer_Cancelled);
   end loop;
end Periodic_Task;
```

Needing task dispatching policy-specific operations has a negative effect on code reuse, since a proven task requires modification and retesting to adapt to different policies.

The amount of work required to build cyclic tasks has been acknowledged by users in the Ada real-time community, resulting in two approaches seeking to make cyclic tasks easier to implement by utilising Ada's library support to provide reusable cyclic task building blocks. The two approaches lean on different data reuse abstractions: the GNAT Component Collection utilises generic packages to contain the common cyclic code, while the *Real-Time Utilities Library* leverages Ada's tagged types.

GNAT Component Collection Ravenscar Patterns

The GNAT Component Collection (GNATcoll) by AdaCore provides an assortment of general-purpose packages used by AdaCore and which are available to end-users (AdaCore, 2013B). For real-time systems, GNATcoll provides what the collection calls multitasking patterns: a set of packages factoring the cyclic task template code into Ravenscar-compliant generic packages. The patterns offer only a subset of the *cyclic task model* — simple periodic and sporadic tasks — in addition to servers that extend the tasks to accept different parameters or requests on each task invocation. Despite the small coverage of the real-time model, they sufficiently show the benefits and limitations of factoring cyclic task template code into generic packages.

```
generic

   Task_Priority     : System.Priority;
   Phase             : Millisecond;
   Period            : Millisecond;
   System_Start_Time : Ada.Real_Time.Time := Ada.Real_Time.Clock;
   with procedure Cyclic_Operation;

package GNATCOLL.Ravenscar.Simple_Cyclic_Task is

private

   task Simple_Cyclic_Task is
      pragma Priority (Task_Priority);
   end Simple_Cyclic_Task;

end GNATCOLL.Ravenscar.Simple_Cyclic_Task;
```

Listing 9 GNATCOLL.Ravenscar.Simple_Cyclic_Task package specification.

```
generic

   Task_Priority             : System.Priority;
   Minimum_Interelease_Time : Millisecond;
   System_Start_Time : Ada.Real_Time.Time := Ada.Real_Time.Clock;
   Protocol_Ceiling  : System.Any_Priority;
   --  the ceiling priority of the protected object used to post
   --  and fetch requests

   with procedure Sporadic_Operation;

package GNATCOLL.Ravenscar.Simple_Sporadic_Task is

   procedure Release;
   --  used by client to trigger the task

private

   protected Protocol is
      pragma Priority (Protocol_Ceiling);

      procedure Release;

      entry Wait (Release_Time : out Ada.Real_Time.Time);

   private
      Barrier : Boolean := False;
      Pending : Integer := 0;
   end Protocol;

   task Simple_Sporadic_Task is
      pragma Priority (Task_Priority);
   end Simple_Sporadic_Task;

end GNATCOLL.Ravenscar.Simple_Sporadic_Task
```

Listing 10 GNATCOLL.Ravenscar.Simple_Sporadic_Task package specification.

The packages GNATCOLL.Ravenscar.Simple_Cyclic_Task (Listing 9) and GNATCOLL.Ravenscar.Simple_Sporadic_Task (Listing 10) provide GNATColl's Ravenscar tasking patterns. At their heart, each generic package contains a task whose task body contains the appropriate cyclic task template, with the task's cyclic attributes passed via the generic package's parameters. The user provides the task's cyclic code in the form of a procedure, also passed through the generic package's parameters. Thus, a simple periodic task would look like:

Listing 11
Simple periodic task using GNATColl

```
procedure Cycle_Operation;

package My_Periodic_Task is
  new GNATCOLL.Ravenscar.Simple_Cyclic_Task
      (Task_Priority     => 10,
       Phase             => 500,
       Period            => 100,
       System_Start_Time => System_Properties.Start_UP_Time,
       Cyclic_Operation  => Cyclic_Operation);

... persistent task state ...

procedure Cycle_Operation is
  ... cycle state ...
begin
   ... cyclic code ...
end Cycle_Operation;
```

While a simple sporadic task:

Listing 12
Simple sporadic task using GNATColl

```
procedure Cycle_Operation;

package My_Sporadic_Task is
  new GNATCOLL.Ravenscar.Simple_Sporadic_Task
    (Task_Priority            => 10,
     Minimum_Interelease_Time => 1_000,
     Protocol_Ceiling         => 15,
     System_Start_Time        => System_Properties.Start_UP_Time,
     Sporadic_Operation       => Cycle_Operation);

... persistent task state ...

procedure Cycle_Operation is
   ... cycle state ...
begin
   ... cyclic code ...
end Cycle_Operation;
```

with another task or interrupt handler raising cyclic events via the My_Sporadic_Task.Release procedure call.

Hiding the cyclic task template within a generic package simplifies the creation and understanding of cyclic tasks in Ada. However, doing so buries Ada's task abstraction inside the generic package. Consequently, the generic library approach looses the advantage Ada's task abstraction provides: collocating a task's state and code in a single programming unit. State persisting across

cycles now lies in a scope not exclusive to the cyclic operations, opening the way for other tasks to incorrectly access the task's state. From a maintenance perspective, without carefully applied coding conventions difficulties lie in quickly determining which package-level objects belong to a task.

Finally, the GNATColl approach imposes its own complexity if a system requires multiple copies of a cyclic task. If the task has a persistent state, then each copy of the task requires a unique set of external state variables. Thus, each task copy requires its own task body procedure referencing the correct state objects. Accordingly, development costs increase because the same task body procedure requires duplication and modification to reference unique state objects, increasing the risk for faults from incorrect modifications. Likewise, maintenance burden increases due to changes to the task body requiring replication across the multiple copies of the task body procedure.

Real-Time Utilities Library

Andy Wellings and Alan Burns (2007) presented an alternative approach to implementing cyclic tasks through a framework of common real-time utilities. These reusable utilities are defined in a standards-like manner, alleviating the work required to build a cyclic task while supporting real-time systems: offering periodic, sporadic and aperiodic tasks; with optional deadline and execution budgets support; and execution servers supporting aperiodic tasks in real-time systems.

At its core, the *Real-Time Utilities Library* breaks a cyclic task into three components:

- A task state object that contains the state of the task and the user code that operates on the state.
- A release mechanism used to release each cycle. This component also detects missed deadlines and execution overruns.
- A real-time task, implementing the cyclic behaviour.

An additional fourth component supports aperiodic tasks sharing common CPU budget.

Each component in the *Library* (Table 1 on page 26) provides a set of types that offer differing behaviour that, once assembled, provides the user a cyclic task with the desired attributes. Underlying this library of real-time components is tagged records and type interfaces, minimising the number of types each component provides. For the task state object, the library provides three abstract tagged records to support three types of cyclic tasks: periodic, sporadic and aperiodic. Users then create their own task state type by extending one of these tagged records to include their own state and actions; all without needing to touch the state associated with the cyclic template. The release mechanism types use protected objects to provide different cycle release behaviours and support for missed deadlines and execution time overruns. Finally, the task itself with its implemented cyclic behaviour is provided through one of several real-time tasks, differing in their termination semantics, and brings together the task, its state and its release mechanism.

Table 1
Real-time components provided by the Real-Time Utilities Library

State Objects *(abstract tagged types)*		
Periodic_Task_State	Sporadic_Task_State	Aperiodic_Task_State

Release Objects *(protected types)*	
Periodic_Release	Periodic_Release_With_Deadline_Miss
Periodic_Release_With_Overrun	Periodic_Release_With_Deadline_Miss_And_ Overrun
Sporadic_Release	Sporadic_Release_With_Deadline_Miss
Spoardic_Release_With_Overrun	Sporadic_Release_With_Deadline_Miss_And_ Overrun
Aperiodic_Release	Aperiodic_Release_With_Deadline_Miss
Aperiodic_Release_With_Overrun	Aperiodic_Release_With_Deadline_Miss_And_ Overrun

Real-Time Tasks *(task types)*	
Simple_Real_Time_Task	Real_Time_Task_With_Deadline_Termination
Real_Time_Task_With_Overrun_ Termination	Real_Time_Task_With_Deadline_And_Overrun_ Termination

Listing 13
Simple sporadic task using Real-Time Utilities Library.

```
type My_Sporadic_Task_State is
  new Sporadic_Task_State with record
   ... task state ...
end record;

procedure Initialize (S : in out Sporadic_Task_State);
procedure Code (S : in out Sporadic_Task_State);
procedure Overrun (S : in out Sporadic_Task_State);

Task_State       : aliased My_Sporadic_Task_State;
Task_Release     : aliased Sporadic_Release
                            (S => Task_State'Access);
My_Sporadic_Task : Real_Time_Task
                      (S => Task_State'Access,
                       R => Task_Release'Access,
                       Init_Prio => Default_Priority);

procedure Initialize (S : in out Sporadic_Task_State) is
begin
   S.Pri := 10;
   S.MIT := Milliseconds (500);
end Initialize;

procedure Code (S : in out Sporadic_Task_State) is
begin
   ... cyclic code ...
end Code;
```

Listing 13 and Listing 14 demonstrate use of the *Real-Time Utilities Library* for a simple sporadic and aperiodic task respectively. A new cycle for both tasks requires a release call to the task's release mechanism object (for the examples this would be `Task_Release.Release`). For the aperiodic task, the *Real-Time Utilities Library* automatically attaches the task to the specified execution server. Since the *Library* targets real-time systems, all aperiodic tasks require attachment to an execution server.

Listing 15 shows a simple periodic task using the *Real-Time Utilities Library*, while Listing 16 extends the task to enforce deadlines and execution budgets. The latter listing demonstrates an appealing aspect of the *Library*: building a complex cyclic task is only slightly more complicated than a simple cyclic task.

Together, these examples highlight the structure the *Real-Time Utilities Library* uses to implement cyclic tasks and how the three *Library* components link together. Together with abstracting the cyclic code, the *Real-Time Utilities Library* provides a clarity and structure that does not exist when implementing cyclic tasks directly with Ada's low-level tasking primitives.

Listing 14
Simple aperiodic task using Real-Time Utilities Library.

```
A_Sporadic_Server : Execution_Server.Sporadic.Sporadic_Server
                      (Period => Milliseconds (500),
                       Budget => Milliseconds (50);
                       Prior  => 10);
--  An implementation defined execution server.

type My_Aperiodic_Task_State is
  new Aperiodic_Task_State with record
    ... task state ...
end record;

procedure Initialise (S : in out Aperiodic_Task_State);
procedure Code (S : in out Aperiodic_Task_State);
procedure Overrun (S : in out Aperiodic_Task_State);

Task_State        : aliased My_Aperiodic_Task_State;
Task_Release      : aliased Aperiodic_Release
                             (S  => Task_State'Access
                              ES => A_Sporadic_Server);

My_Aperiodic_Task : Real_Time_Task
                     (S => Task_State'Access,
                      R => Task_Release'Access,
                      Init_Prio => Default_Priority);

procedure Initialize (S : in out Aperiodic_Task_State) is
begin
   S.Pri := 01;
end Initialize;

procedure Code (S : in out Aperiodic_Task_State) is
begin
   ... cyclic code ...
end Code;
```

Listing 15 Simple periodic task using Real-Time Utilities Library.

```
type My_Periodic_Task_State is
  new Periodic_Task_State with record
   ... task state ...
end record;

procedure Initialize (S : in out Periodic_Task_State);
procedure Code (S : in out Periodic_Task_State);
procedure Overrun (S : in out Periodic_Task_State);

Task_State       : aliased My_Periodic_Task_State;
Task_Release     : aliased Periodic_Release
                             (S => Task_State'Access);
My_Periodic_Task : Real_Time_Task
                     (S => Task_State'Access,
                      R => Task_Release'Access,
                      Init_Prio => Default_Priority);

procedure Initialize (S : in out Periodic_Task_State) is
begin
   S.Pri    := 10;
   S.Period := Milliseconds (500);
end Initialize;

procedure Code (S : in out Periodic_Task_State) is
begin
   ... cyclic code ...
end Code;
```

Despite these observations, the *Real-Time Utilities Library* approach falls short of meeting the needs of real-time users:

Loss of task abstraction The *Real-Time Utilities Library* loses Ada's task abstraction because the task type only implements the cyclic behaviour. Declaration of a task's state occurs outside the task body, allowing another task within the same scope to read and modify the former task's state. Non-state task-specific declarations have to be defined by repeating the declarations in both the initialisation and code procedures or by exposing these declarations in a scope outside the task — approaches not supporting the development of maintainable software.

Decomposing the cyclic task abstraction The *Real-Time Utilities Library* approach requires the user to think of cyclic tasks as three separate objects, even though conceptually a cyclic task is a single object. Thus, the *Library* still requires the user to use a lower programming abstraction than what they have designed around. An implication of this is a simple periodic task, with no deadline or execution time monitoring, is more complex to write using the Library.

The specification of a cyclic task is not clear Recreating the specification of a cyclic task from its Real-Time Utilities implementation is not straightforward. Discovery of the type of cyclic task implemented requires the reader to examine what type the state was derived from or examine the release mechanism used. Other cyclic task attributes require the reader to delve into the body of the initialisation procedure to discover their values. This is problematic as it mixes the specification of the task with the code initialising the task's user

```
type My_Periodic_Task_State is
   new Periodic_Task_State with record
   ... task state ...
end record;

procedure Initialize (S : in out Periodic_Task_State);
procedure Code (S : in out Periodic_Task_State);
procedure Overrun (S : in out Periodic_Task_State);

Task_State       : aliased My_Periodic_Task_State;
Task_Release     : aliased
                 Periodic_Release_With_Deadline_Miss_And_Overrun
                   (S => Task_State'Access,
                    Termination => True);
My_Periodic_Task : Real_Time_Task_With_Deadline_Termination
                     (S => Task_State'Access,
                      R => Task_Release'Access,
                      Init_Prio => Default_Priority);

procedure Initialize (S : in out Periodic_Task_State) is
begin
   S.Pri               := 10;
   S.Period            := Milliseconds (500);
   S.Relative_Deadline := Milliseconds (400);
   S.Execution_Time    := Milliseconds (100);
end Initialize;

procedure Code (S : in out Periodic_Task_State) is
begin
   ... cyclic code ...
end Code;

procedure Overrun (S : in out Periodic_Task_State) is
begin
   ... handle exhausted execution budget ...
end Overrun;
```

Listing 16
A periodic task with deadlines and execution budgets using the Real-Time Utilities Library. A missed deadline causes the task to terminate, while execution budget exhaustion invokes a handler procedure.

state, harming readability, and requires the procedures of the task to divine its timing attributes.

Cannot make cyclic task properties static Schedulability analysis for hard real-time systems is generally only feasible with static cyclic attributes. Since the *Real-Time Utilities Library* stores the cyclic task attributes inside the same state object as the user's task state, it is not possible for the compiler to enforce this constraint; thus allowing these attributes to change at run-time, leading to maintenance and reliability problems.

Unusable on some systems The extensive use of Ada's tasking and object-oriented features prevents many high-integrity systems from employing the *Library* (such as systems using the Ravenscar Profile). Specifically, the prohibition of dynamic dispatching precludes the Library's use in these systems. Additionally, its release mechanisms rely on protected objects with multiple entries, the ability to requeue entry calls and the use of execution time timers — all parts of the Ada tasking run-time forbidden under the Ravenscar Profile.

Prior Work Incorporating Cyclic Tasks into Ada

Incorporating a cyclic task model into Ada is, in and of itself, not an original concept, with its origins lying in the mismatch between Ada's original design and the needs of a subset of its users: real-time developers. Some of the underlying real-time concepts were originally described within the specification documents guiding the development of Ada (HOLWG, 1977; 1978) and the surrounding commentary (Fisher, 1978) — with real-time systems emphasised as a target of the language. However, when initially standardised as ANSI/MIL-STD-1815A Ada employed an event-driven tasking model (Baker & Shaw, 1988), with the language subsequently criticised for its inability to support real-time applications despite the original goals (see Alvarez, 1987; Burns & Wellings, 1987; 1988; Cornhilll, Sha, Lehoczky, Rajkumar, & Tokuda, 1987; Maule, 1986; McCormick & McCormick, 1987).

Initially, real-time support for Ada 83 had to come through interfaces to the underlying run-time. The Ada community, through SIGAda's *Ada Run-Time Environment Working Group* (ARTEWG), developed a common interface called *The Catalogue of Interface Features and Options* (CIFO): providing commonly required Ada run-time environment features not covered by the language itself (Ada Runtime Enviroment Working Group, 1991). Real-time systems were a particular focus of CIFO, with the interface incorporating a comprehensive scheduling model supporting a form of cyclic tasks.

Language support for real-time systems appeared in the subsequent language revision, Ada 95 (Barnes, 1995; ISO/IEC, 1995), with further revisions incrementally building the extensive real-time support that appears in Ada today. Ada 95 largely replaced the functionality of CIFO with low-level building blocks, with the expectation users would replicate the high-level interfaces of CIFO using these blocks (Tokar, 1992). This included CIFO's cyclic task behaviour despite suggestions otherwise.

Ted Baker (1989) argued the inclusion of a periodic task model with deadline enforcement was a simpler and more elegant solution than a language-defined scheduling package; additionally it would resolve Ada 83's inadequate `Calandar.Clock` and delay facilities. Baker's approach pushed the periodic release of a task into Ada's tasking model by defining new attributes and representation clauses for the properties of a periodic task. The approach still used a *wait for event statement* — a repurposed parameterless **delay** statement — but relied on the run-time to calculate when to release the task:

Listing 17
An example of Ted Baker's (1989) solution for incorporating a periodic task abstraction into Ada.

```
task body T is
   for T'Period use 0.001;
   for T'Deadline use 0.005;
begin
   loop
      -- cycle statements
      delay;
   end loop;
end T;
```

Lee Lucas, on the other hand, submitted a different approach as a *Revision Request* to the Ada 9X Project, focusing on simple periodic tasks (Ada 9X Pro-

ject Office, 1990B, pp. 9-127–128). In addition to the existing concerns with the then current Ada approach of using a loop with a relative delay statement, Lucas stated that such

> *expression of periodic tasks in this manner is idiomatic (and idiosyncratic?) making it more likely that logic errors will be made when programming periodic tasks.*
>
> — *Lee Lucas (Ada 9x Project Office, 1990B)*

Lucas used a similar approach to Baker, repurposing the **delay** statement to implement a *wait for event statement* within the task body while using a new pramga Periodic to define the periodic task's period attribute:

```
task body T is
   pragma Periodic (Task_Period);
begin
   loop
      -- cycle statements
      delay;
   end loop;
end T;
```

Listing 18
An example of Lee Lucas' (Ada 9X Project Office, 1990B) solution to incorporating a periodic task abstraction into Ada.

Lucas also suggested his approach could similarly provide sporadic and aperiodic tasks by defining Sporadic and Aperiodic pragmas.

Since the development of Ada 95, suggestions for incorporating a cyclic task-like model into Ada shifted from the language's task abstraction towards its scheduler model (see Aldea, Miranda, & Harbour, 2004; Rivas & Harbour, 2003; Rivas, Miranda, & Harbour, 2005) and then full circle back to a library providing cyclic task patterns (Wellings & Burns, 2007).

Two reasons exist for the aversion to support a cyclic task abstraction directly in Ada's syntax:

- the perceived added complexity to the language; and
- high-level abstractions struggle to meet the needs of a wide set of users.

Lucas' *Revision Request* was rejected by the Ada 9x Requirements Team for the first reason (Ada 9x Project Office, 1990A). While considered desirable,

> *a generalized specification of 'periodic' would generate a complete new task class within Ada. Its behaviour in rendezvous, priority, and possible restrictions make this language change expensive. Since all of the required functionality can be provided with [delay until statement], it is judged out-of-scope.*
>
> — *Ada 9x Project Office, 1990A*

The second reason came from the Ada 83 experience: many of its high-level language abstractions did not meet the needs of users and hindered the implementation other abstractions (Wellings & Burns, 2007). For real-time users, the biggest example was the unsuitability of the event-driven tasking model. Likewise, the periodic task abstractions were felt by many to be too restrictive and lacking the functionality users would desire (Tokar & Wellings, 2003).

Cyclic Tasks in Other Languages

While Ada is unique as a language designed to meet the software engineering needs of embedded systems, other languages and language extensions have been developed that attempt to address the technical needs of real-time systems (Stoyenko, 1992). While frequently these languages are only concerned about access to device memory and real-time clocks, many include explicit support for cyclic tasks though a variety of means: libraries, temporal scopes and cyclic task syntax.

Library-Based Cyclic Tasks

Language-defined libraries provide the most common route to incorporate cyclic task behaviour in a language. As CIFO and the Real-Time Utilities Library demonstrated, libraries can provide the semantics for cyclic tasks through a collection of run-time procedure calls or objects. Industrial Real-Time FORTRAN (Kneis, 1981) is an early example of this approach: defining a set of run-time calls to schedule different kinds of cyclic task and to connect the running of tasks to events.

By contrast, the Real-Time Specification for Java (RTSJ) (Bollella et al., 2000; Hunt et al., 2017) provides a defined set of connected objects that come together to create some cyclic tasks and whose design is the foundation for the Real-Time Utilities Library. As a newer language extension under active development since the 1990's, RTJS facilitates the creation of a rich and diverse set of cyclic task behaviours, including the ability to queue aperiodic events, and the activation and deactivation of periodic tasks by external events.

However, as discussed in this chapter these library-based approaches impede the development of reliable and maintainable software. They cannot offer their languages the task abstraction Ada provides: a scope for the state of the task, and the clear identification of a task, its state and code. Furthermore, relying on run-time calls to implement cyclic task behaviour mixes the code making a task cyclic with the cyclic activity the task performs: making it convoluted to read and write cyclic tasks, particularly in understanding their properties.

On the other hand, library-define objects like those defined in Real-Time Utilities Library and RTSJ require the decomposition of the cyclic task abstraction. Building cyclic tasks this way asks the user to perform more work as they need to decompose their cyclic task and determine which objects are required to get the desired behaviour. Additionally, object-based approaches complicate efforts to understand the cyclic behaviour of the task as the different pieces of the cyclic task, which together increase the risk for programming and comprehension errors. RTSJ is also hampered by having to fit within the threading model of Java which possesses an object-oriented world view: to complete the cyclic task the user must implement a loop within the object thread with an explicit run-time call at the end of the loop. A simple example of RTSJ and its readability is demonstrated by the simple periodic task in Listing 19.

```
public class PeriodicCounter extends RealTimeThread {
   private int count = 0;

   public void run() {
      while(true) {
         System.out.println("Count " + count);
         waitForNextPeriod();
      }
   }

   public static void main(String[] args) {
      PriorityParameters taskPrior = new PriorityParameters(20);
      PeriodicParameters taskParm = new PeriodicParameters(
         new RelativeTime(500, 0), // Phase
         new RelativeTime(500, 0), // Period
         new RelativeTime(100, 0), // Execution budget
         new RelativeTime(300, 0), // Deadline,
         null, null); // Execution and deadline handlers
      PeriodicCounter task = new
        PeriodicCounter(taskPrior, taskParm);
   }
}
```

Listing 19
A periodic task using the Real-Time Specification for Java.

Temporal Scope-Based Cyclic Tasks

DPS (Lee & Gehlot, 1985) introduced the novel concept of temporal scopes to support the timing specification for distributed real-time software. Like block statements in Ada providing a scope for objects, DPS provides block statements that allow users to specify the timing constraints for the code executed within the block together with timing-related exception handlers for when the constraints are violated. DPS is notably the first language to allow the specification and enforcement of absolute and relative deadlines and execution budgets.

The language defines three different temporal scopes: local, repetitive and consecutive. The former two temporal scopes facilitate cyclic tasks construction: aperiodic and sporadic task buildable from local temporal scopes and DPS's inter-task communication, and periodic tasks from repetitive temporal scopes.

DPS defines temporal scopes and their properties through its syntax. As shown in Listing 20, local temporal scopes use keywords to identify a range of different properties that offer flexibility to define timing specifications based on relative or absolute times. Furthermore, execution_part can restrict the running time of the scope to either a set quantity of processor time or an elapse time since the scope began executing. Similar syntax provides repetitive temporal scopes.

Listing 21 demonstrates the relative ease of building cyclic tasks using temporal scopes, particularly as the language takes care of enforcing the rich temporal constraints which the user would have to build themselves in Ada. However, temporal scopes bury the cyclic behaviour of the task inside its implementation, making it harder to identify the cyclic task and its properties. Thus, temporal scopes decouple the cyclic task abstraction: only a part of their execution is cyclic, not the task itself.

Listing 20
DPS language specification for local and repetitive temporal scopes.

```
local_temporal_scope ::=
   start delay_part [execution_part] [deadline_part] do
      statements
   [exception
      exception  handlers]
   end;

delay_part ::= now | at absolute_time | after relative_time
execution_part ::= execute relative_time | elapse relative_time
deadline_part ::= by absolute _time | within relative_time

repetitive_temproal_scope ::=
   from start_time to end_time every period execute execute_time
     within deadline do
      statements
   [exception
      exception_handlers]
   end;

start_time, end_time ::= absolute_time
period, execute_time, deadline ::= relative_time
```

Listing 21
Periodic and sporadic counters written in DSP.

```
process periodic_counter;
   var Count : Integer;
begin
   from now to now + 50 min every 10 sec execute 3 sec
     within 5 sec do
      Count := Count + 1;
   end;
end;

process sporadic_counter;
   call-port Trigger;
   var Count   : Integer;
   var Release : Time
begin
   Release := now + 20 min
   while true do
      receive (Trigger, nil, 1);
      start at Release execute 1 min within 10 min do
         Count := Count + 1;
      end;
      Release := Release + 20 min
   end;
end;
```

DPS also makes extensive use of keywords to delimit temporal scope properties. While for a new language this can seem acceptable, it limits the ability for the language to expand with new properties in the future because picking any new keyword will impact existing software. For the same reason, the approach used by DPS to use keywords would be difficult to apply to an existing language as these new keywords will conflict with existing code written in the base language.

Syntax-Based Cyclic Tasks

Most languages and language extensions incorporating support for cyclic tasks have seen little or no use. HAL/S (Newbold et al., 1976) and PEARL (DIN, 1998) form two exceptions. While developed independently in USA and Germany respectively, both languages share similar characteristics: developed in the early 1970's by industrial users based on PL/I and include dedicated tasking syntax.

Intermetrics developed HAL/S as a real-time aerospace programming language for NASA's Space Shuttle and found use in other aerospace projects like the Global Positioning System, European Space Agency's Spacelab and the Galileo space probe before being displaced by Ada (Klumpp, 1985). By contrast, a collaboration of German industry and universities developed PEARL for the control of real-time processes used in industries and laboratories, which was standardised by the German standards body DIN first in 1981 and most recently in 1998 (GI-Working Group 4.4.2, 1998; Martin, 1978). Many German industry applications still use PEARL today.

HAL/S

HAL/S considers real-time statements as integral to the language, with the view that "real-time constraints are an intrinsic part of the application" and such statements "allow the programmer to express the entire algorithm directly and in one place." (Ryer, 1979). The language also considers the portability of language syntax over operating system calls as a benefit.

As Listing 22 on page 36 shows, HAL/S provides a task abstraction similar to Ada: a scope collecting the state of a task and its operations together. Differing from Ada, HAL/S treats tasks like procedures rather than types. Consequently, tasks are created through the SCHEDULE statement, which offers the ability to create periodic tasks. The SCHEDULE statement offers a degree of flexibility to scheduler tasks, with the start of the task deferrable till an absolute or relative time, or an event occurring. HAL/S can schedule a periodic task to run at periodic intervals or after a fixed inter-task period. The language can also cancel the periodic task after a set period or in response to event variables changing[1].

While providing a powerful way to schedule periodic and repetitive tasks, and to activate/deactivate periodic tasks in response to event changes, HAL/S does not provide support for aperiodic and sporadic tasks in its schedule statement. Once cancelled or completed, a task can only restart in response to another schedule statement. Consequently, aperiodic and sporadic tasks must be built in a similar manner to Ada. Finally, HAL/S does not support deadlines, while using statements to schedule tasks creates difficulties in locating cyclic task properties because this approach separates task cyclic properties from the tasks.

PEARL

PEARL offers a similar set of capabilities and syntax to HAL/S as demonstrated in Listing 23 on page 36. The language treats tasks like procedures and provides the ability to schedule tasks through the ACTIVATE statement. Like the SCHEDULE statement in HAL/S, the ACTIVATE statement can provide an optional start condition for the task — either a time or interrupt — and allows the scheduling of periodic tasks with fixed periods and optional deactivation times.

1. Event variables include variables of type EVENT and the running state of other tasks, and allows Boolean combination of events.

Listing 22
Cyclic tasks written in HAL/S.

```
CYCLIC_HEAVEN:
PROGRAM:
   DECLARE TRIGGER EVENT;

PERIODIC_TASK:
TASK;
   ... periodic code ...
CLOSE PERIODIC_TASK;

COUNTER:
TASK;
   ... counter code ...
CLOSE COUNTER;

ONE_SHOT:
TASK;
   ... one shot code ...
CLOSE ONE_SHOT;

   SCHEDULE PERIODIC_TASK IN 10.5 PRIOIRTY (50), REPEAT EVERY 0.1
            UNTIL RUNTIME + 1000;
   SCHEDULE COUNTER AT 1000 PRIORITY (40), REPEAT AFTER 0.05
            WHILE PERIODIC_TASK;
   SCHEDULE ONE_SHOT ON TRIGGER;
CLOSE STARTUP;
```

Listing 23
Cyclic tasks written in PEARL.

```
MODULE;
SYSTEM;
   Trigger: Hard_Int (2);

PROBLEM;
SPECIFY Trigger INTERRUPT;

Main_Task: TASK MAIN PRIORITY 2;
           AFTER 30 SEC ALL 10 SEC DURING 2 HRS
             ACTIVATE Periodic_Task PRIORITY (10);
           AT 10:00 ALL 50 SEC ACTIVATE Counter;
           WHEN Trigger ACTIVATE Aperoidic_Task
           END; ! Main_Task

Periodic_Task:  TASK;
               . .. periodic code ...
                END; ! Periodic_Task

Counter:        TASK;
                ... counter code ...
                END; ! Counter

Aperoidic_Task: TASK;
                ... aperiodic code ...
                END; ! Aperoidic_Task
```

Compared to HAL/S, PEARL provides richer semantics for cyclic tasks. A second ACTIVATE statement can reschedule a periodic task, while tasks with interrupt-based start conditions are scheduled with each raising of that interrupt. Thus, PEARL provides support for both periodic and aperiodic tasks, though like HAL/S does not offer language support for sporadic tasks or deadline conditions, and suffers from the properties of cyclic tasks being located within the body of the program.

Real-Time Euclid

Real-Time Euclid was a novel language designed to support schedulability analysis for real-time systems (Kligerman & Stoyenko, 1986). While never seeing outside use, its focus on schedulability saw the language define cyclic task properties on task declarations to allow schedulability analyser tools to identify these properties (Stoyenko, 1992). Furthermore, Real-Time Euclid only supports periodic and sporadic tasks.

```
var CyclicMayhem: module
   var Trigger: activation condition

   process PeriodicTask: periodic frame 30
     first activation atTime 100
      ... periodic code ...
   end PeriodicTask

   process Counter: periodic frame 100
      ... counter code ...
   end Counter

   process SporadicTask: atEvent Trigger frame 500
      ... sporadic code ...
   end SporadicTask
end CyclicMayhem
```

Listing 24
Cyclic tasks written in Real-Time Euclid.

As demonstrated in Listing x, Real-Time Euclid supports activating a task periodically or in response to an interrupt. For periodic tasks, the first activation can be a time, the raising of an interrupt or the earliest of either two events. Unlike HAL/S and PEARL, the cyclic specification of the task does not provide a means to deactivate a periodic task after a set period. However, Real-Time Euclid does support a limited means of deadline enforcement, with the language designed to raise a run-time error if a task exceeds its compiler calculated deadline (made possible by the constructs of the language being time bounded). Once again, the use of keywords to specify cyclic task parameters makes it difficult to add new parameters without affecting existing users.

Conclusion

Cyclic tasks form the cornerstone of real-time systems and derive from the nature of the fundamental system they are apart of: control systems — systems which manipulate their environment in response to stimuli in a calculated manner. In turn, the ultimately periodic nature of real-time tasks enables system designers to determine the schedulability of the system before its operation. This valuable technique helps identify and resolve timing faults early in a systems development, and frees users from trying to locate and resolve otherwise costly defects. Ultimately, a schedulability analysis enables developers to guarantee a system is free of timing defects, providing the necessary assurance for safety-critical applications.

Constraining the classical real-time task model has created a general cyclic task model to support the development of reliable and maintainable real-time systems free of timing defects. The model centres on tasks as the basic unit of work: an execution unit encapsulating the code, state and execution of an activity or algorithm. This definition fits neatly into the same definition used by Ada. The difference between Ada and the cyclic task model lies with Ada only offering direct support for sequential tasks, whereas the model introduces the concept of cyclic tasks: tasks' whose code executes in response to an event.

By contrast, Ada tries to replicate the functionality of the cyclic task model through its existing tasking model, supported by the extensive library services provided by its *Real-Time System Annex*. Ada 2012 achieves this with success, being able to comprehensively cover the model with a set of lower-level real-time primitives, offering the ability to: build periodic, sporadic and aperiodic tasks; support cycle deadlines and execution budgets; build execution servers; and support a wide variety of schedulers. By providing a set of building blocks from which to build cyclic tasks, Ada offers users the ability to build the cyclic task that meets their needs.

However, this Chapter has shown Ada's approach requires considerable and repetitive effort, increasing the scope for faults to enter a system. Ada's prized traits of readability and maintainability suffer, with cyclic task attributes hard to identify from a system's code and requires access to the body of the task. Compounding the problem, clear separation of a cyclic task's code and the parts of the task making it cyclic does not exist.

The cost of building cyclic tasks on Ada would not be an issue if it were not for the importance of the abstraction for real-time systems. Two noted efforts exist in response — GNATColl's Ravenscar patterns and the *Real-Time Utilities Library* — simplifying cyclic tasks by abstracting much of the cyclic task template into reusable libraries. While these library-based solutions do reduce the effort and the amount of code required to implement a cyclic task, they do so by losing Ada's task abstraction.

Losing Ada's task abstraction negatively affects the development of real-time systems, as the abstraction provides a scope (enforced by the compiler) which only the task can access. The scope protects the integrity of a task by preventing users from incorrectly accessing the task's state and clearly identifies to the user the task, its state and the code that operates on the state.

Finally, even comprehensive solutions like the *Real-Time Utilities Library* requires decomposition of the cyclic task abstraction, leaving extra work for users

to write and understand cyclic tasks. Furthermore, Ada's real-time facilities are not task dispatching policy independent, reducing the ability to seamlessly reuse existing tasks across different systems.

Consequently, while Ada provides the building blocks to build real-time cyclic tasks, the act of building them imposes development overheads: affecting the reliability, maintainability and cost of real-time systems. Because cyclic tasks form the cornerstone of real-time systems, the lack of a cyclic task model hampers Ada's ability to lower software lifecycle cost.

As this Chapter detailed, many of these concerns were raised as part of the Ada 9X review process where suggestions were put forward to incorporate cyclic tasks into the language. While dismissed at the time for being too high-level and increasing the complexity of the language, this Chapter has shown other languages targeting real-time systems have successfully included cyclic task models in their language while also demonstrating the flaws with approaches that use library-based solutions.

HAL/S and PEARL have shown syntax-based cyclic tasks can have success in industrial applications, with PEARL actively used today. Their success derives from being developed on behalf of users rather than as an academic project. While later real-time languages like Real-Time Euclid and DSP have not had any industrial success, they have demonstrated favourable cyclic task features. Real-Time Euclid established the advantage of having cyclic task properties on the definition of a task, while DSP showed the possibility of providing a rich real-time model allowing the description and enforcement of execution budgets and deadlines.

The Chapter also demonstrated the main flaw with syntax-based approaches: relying on syntax to annotate a task with its cyclic task properties. Incorporating a similar approach in Ada would cause difficulties as these keywords will already be used by users as identifiers. The earlier suggestions for cyclic tasks within Ada's semantics provides a path forward: using pragma's or attributes to annotate the cyclic properties of a task without relying on new syntax.

QANTAS
Spirit of Australia
VH-VXC
3

Chapter 3

The Cyclic Task Specification

This chapter presents the dissertation's new cyclic task abstraction for Ada. The *Cyclic Task Specification* aims to preserve Ada's existing task abstraction while providing a cyclic task abstraction that considers a cyclic task as a single object. In doing so, the dissertation provides the key real-time abstraction in a way which resolves the problems identified with existing cyclic task implementations while fitting within the existing language.

Minimising language impact, the *Specification* provides cyclic tasks through a new optional cyclic section inside the body of an Ada task. Semantics of an Ada task only change when this cyclic section is present, with its behaviour tailored through a set of new task attributes, set dynamically via run-time procedures or statically using Ada 2012's aspects. Interaction of Ada tasks — with or without the cyclic section — with the rest of the language remains unchanged.

This chapter opens by introducing the *Cyclic Task Specification*. Examples of the *Specification*'s usage and implementation follows, presenting the benefits incorporating a cyclic task abstraction brings to the reliability and maintainability of real-time systems. Finally, the chapter will explore the impact the *Specification* has on Ada and how it resolves issues that prevented the incorporation of prior periodic abstractions.

The Specification

The *Cyclic Task Specification* modifies Ada to incorporate a cyclic task abstraction within Ada's broader task abstraction. Working within and leveraging the existing language forms a key design feature of the *Specification*, minimising the introduced complexity. Consequently, the *Specification* does not introduce a new type for cyclic tasks.

Instead, the existing sequential task type (ARM 9.1) generalises to become a task type encompassing both sequential and cyclic task abstractions. The changes the *Cyclic Task Specification* makes to Ada breaks down into four areas: task syntax, task attributes, task model and execution servers.

Task Syntax

Of Ada's existing tasking syntax, only the task body changes. The new task body provides the general task abstraction: a programming unit capturing the state, operations and execution of a task. It sees the sequential handled_sequence_of_statements replaced with the generalised task_sequence_of_statements. Consequently, Ada's existing task body (ARM 9.1):

```
task_body ::=
   task body defining_identifier [aspect_specification] is
     declarative_part
   begin
     handled_sequence_of_statements
   end [task_identifier];
```

becomes:

```
task_body ::=
   task body defining_identifier [aspect_specification] is
     declarative_part
   begin
     task_sequence_of_statements
   end [task_identifier];

task_sequence_of_statements ::=
    sequential_sequence_of_statements
   [cycle
     cyclic_sequence_of_statements]
   [exception
     exception_handler
    {exception_handler}]

sequential_sequence_of_statements ::= sequence_of_statements
cyclic_sequence_of_statements ::= sequence_of_statements
```

The task_sequence_of_statements captures both the sequential and cyclic statements of the task, with the sequential_sequence_of_statements forming the task's sequential section. Following the new keyword **cycle** lies the cyclic_sequence_of_statements: the task's cyclic section. Both statement sets rename Ada's existing sequence_of_statements to make the task_sequence_of_statements syntax clearer and to help compilers and ASIS users differentiate

between the sets of statements. The exception handlers respond to exceptions raised by either sequence_of_statements.

A sequential task includes only the sequential_sequence_of_statements. Accordingly, the syntax and semantics of an existing Ada task does not change with the introduction of the *Specification*. A cyclic task uses both sets of sequence_of_statements, with the sequential_sequence_of_statements available to users to initialise the task before its cyclic execution. The initialisation section enables simpler cyclic tasks by removing the need to contort the task's initialisation code into the task's cyclic section.

Introducing a new keyword, **cycle**, over reusing the similar but existing keyword **loop** removed the ambiguity **loop** would have caused. From the task body alone, a **loop** keyword with no corresponding **end loop** could mean the task is a cyclic task or it could mean the user made a coding mistake, meaning to write a loop_statement and forgot to write the terminating **end loop**. Similar ambiguity exists if **end loop** is present. Since a loop_statement and the cyclic task statements have different semantics, a new keyword ensures the intention of the task body is clear.

Also added to the core language is a new statement based on the new keyword to flag that a cycle event has occurred for the listed tasks:

```
new_cycle_statement ::= new cycle task_name {, task_name};
```

Task Attributes

Ada defines one task attribute in the core language and a further four in the *Real-Time Systems Annex*:

- Storage_Size (ARM 13.3)
- Priority (ARM D.3)
- CPU (ARM D.16)
- Dispatching_Domain (ARM D.16.1)
- Absolute_Deadline (ARM D.2.6)

Twelve new task attributes and a system property are introduced to support the cyclic task abstraction, with attributes readable via attribute references. The package Ada.Cyclic_Tasks defines the types used by the new attributes. Representation aspects of the same name may be specified on the task type (including the anonymous type of a single task declaration) to set these attributes. Attributes (excluding Cycle_Behavior) can also be set dynamically at run-time using the procedures supplied by the Ada.Cyclic_Tasks.Dynamics package. Separating the dynamic procedures enables static cyclic attributes via:

```
pragma Restrictions (No_Dependence => Ada.Cyclic_Tasks.Dynamics)
```

Applying the restriction has the advantage of forcing the timing specification of a task to be specified on the task's declaration and not inside its body — making the timing specification clear in the user's code and simplifying schedulability analysis. Appendix A provides the specification for each package.

Many of these attributes are valid only for a particular kind of cyclic task. Hence, it is erroneous to specify the corresponding aspect if the attribute is not valid for that kind of task. Furthermore, the run-time raises Tasking_Error for any attribute operation not valid for the target task.

cyclic execution start time

Release of the first cycle for all tasks will not occur before the system defined property Cyclic_Execution_Start_Time. This time provides a common start time for all tasks, allowing cyclic tasks to complete their initialisation before the system commences its cyclic execution. A configuration pragma Cyclic_Execution_Start_Offset delays the start time by the provided amount if an application needs more time to initialise the system:

```
pragma Cyclic_Execution_Start_Offset (Ada.Real_Time.Time_Span);
```

remark In many cases it would make sense for the common start time to be the time when all cyclic tasks have finished their initialisation. However, it is hard to provide a generalised solution which would work for applications where the quantity of cyclic tasks remains unknown until run-time. Without knowing the number of cyclic tasks at compile time, the only solution would be to wait for currently initialising tasks to complete. However, the possibility exists for a set of initialising tasks to all complete before the creation of further cyclic tasks, leading the system to commence cyclic execution earlier than expected.

cyclic task

Attribute:	**T'Cyclic_Task**
Aspect:	Cyclic_Task
Type:	Ada.Cyclic_Tasks.Task_Kind
Type Definition	Task_Kind **is** (Yielding, Periodic, Aperiodic, Sporadic, False);
Default:	False
Valid	any task type, read-only at run-time

Defines whether a task is cyclic and, if so, the type of cycle event which triggers the execution of cycle. If a task has cyclic statements, it shall have a non-False Cyclic_Task aspect on the task type. Conversely, a task without cyclic statements shall only have a Cyclic_Task aspect if it is False or not have the aspect specified at all. Any other scenario is erroneous.

The Task_Kind type describes the five different kinds of cyclic tasks available:

Yielding The task yields the processor before commencing a new cycle.

Periodic The run-time generates cycle events for the task at fixed periodic intervals, determined by the task's Cycle_Period attribute.

Aperiodic Cycle events raised by `new_cycle_statement`.

Sporadic The `new_cycle_statement` raises a cycle event, but not within the interval defined by the time of the last cycle event and the task's Cycle_MIT attribute.

False Indicates the task is a sequential task. While the absence of the Cyclic_Task aspect would be sufficient to indicate a sequential task, providing False enables users to call T'Cyclic_Task on any task.

Unlike other task attributes, the Cycle_Task attribute is not modifiable at run-time. Consequently, the compiler or run-time may use different data-structures to implement the different task kinds.

cycle phase

Attribute:	**T'Cycle_Phase**
Aspect:	Cycle_Phase
Type:	Ada.Real_Time.Time_Span
Default:	Ada.Real_Time.Time_Span_Zero
Valid:	T'Cyclic_Task /= False, Time_Span >= Time_Span_Zero

The non-negative offset added to Cyclic_Execution_Start_Time, resulting in the earliest time the first cycle event for the task may be raised. For periodic tasks, the first cycle event shall be raised at this time.

cycle period

Attribute:	**T'Cycle_Period**
Aspect:	Cycle_Period
Type:	Ada.Real_Time.Time_Span
Default:	Ada.Real_Time.Time_Span_Last
Valid:	T'Cyclic_Task = Periodic, Time_Span >= Time_Span_Zero

The non-negative timespan between successive run-time generated cycle events. The default value permits postponing the setting of the attribute until run-time. If the restriction No_Dependence => Ada.Cyclic_Tasks.Dynamics is present, then the Cycle_Period aspect shall be specified on the type.

cycle minimum inter-arrival time

Attribute:	**T'Cycle_MIT**
Aspect:	Cycle_MIT
Type:	Ada.Real_Time.Time_Span
Default:	Ada.Real_Time.Time_Span_Last
Valid:	T'Cyclic_Task = Sporadic, Time_Span >= Time_Span_Zero

The interval from the last cycle event that a subsequent cycle event shall not be raised in. The default value permits postponing the setting of the attribute until run-time. If the restriction No_Dependence => Ada.Cyclic_Tasks.Dynamics is present, then the Cycle_MIT aspect shall be specified on the type.

early event response

Attribute:	**T'Early_Event_Response**
Aspect:	Early_Event_Response
Type:	Ada.Cyclic_Tasks.Early_Event_Responses
Type Definition	Early_Event_Responses **is** (Raise_Exception, Delay, Ignore);
Default:	Raise_Exception
Valid:	T'Cyclic_Task = Sporadic | Aperiodic

The attribute determines the run-time response to an event being raised inside a sporadic task's MIT interval or while an aperiodic or sporadic task is currently executing a cycle:

Raise_Exception The new_cycle_statement raises Tasking_Error.

Ignore The event is silently ignored.

Delay The event is delayed until the end of the sporadic task's MIT or the cycle completes, whichever comes latter. If a task has an existing delayed event, any further cycle events are dropped.

relative deadline

Attribute:	**T'Relative_Deadline**
Aspect:	Relative_Deadline
Type:	Ada.Real_Time.Time_Span
Default:	Ada.Real_Time.Time_Span_Last
Valid:	any task type, Time_Span >= Time_Span_Zero

The interval a cycle has to complete in from the time of the cycle's event. A value of Time_Span_Last indicates the task does not have a relative deadline. For sequential tasks, the attribute sets the initial absolute deadline for the task based on when the task finishes activating.

absolute deadline

Attribute:	**T'Absolute_Deadline**
Type:	Ada.Real_Time.Time
Valid:	any task type

For cyclic tasks, the time the current cycle shall complete by. For sequential tasks, the time the task shall complete or update the attribute by.

execution budget

Attribute:	**T'Execution_Budget**
Aspect:	Execution_Budget
Type:	Ada.Real_Time.Time_Span
Default:	Ada.Real_Time.Time_Span_Last
Valid:	any task type, Time_Span >= Time_Span_Zero

The amount of execution time a cycle or sequential task may run for. A task is said to consume its execution budget. A value of Time_Span_Last indicates the task has no execution budget and can consume unlimited processor time.

remaining execution budget

Attribute:	**T'Remaining_Execution_Budget**
Type:	Ada.Real_Time.Time_Span
Valid:	any task type

Returns the amount of execution budget a sequential task or cycle has left.

deadline response
budget response

Attribute:	**T'Deadline_Response, T'Budget_Response**
Aspect:	Deadline_Response, Budget_Reponse
Type:	Ada.Cyclic_Tasks.Violation_Response
Type Definition	Violation_Response **is** (Ignore, Handler, Abort_Cycle, Abort_Task);
Default:	Ignore
Valid:	any task type

The run-time responds to two different timing violations: missed deadlines and execution budget exhaustions. The Deadline_Response and Budget_Response attributes set how the run-time responds to these timing violations respectively. A common type — Timing_Violation_Responses — provides four responses:

Ignore The run-time ignores the timing violation.

Handler The run-time calls the protected procedure handler specified by the corresponding Deadline_Handler or Budget_Handler attribute.

Abort_Cycle For a cyclic task, the current cycle is aborted using Ada's *Abort of a Sequence of Statement* semantics (ARM 9.8) and the task waits for the next cycle event. Not valid for sequential tasks.

Abort_Task The task is aborted with Ada's *Abort of a Task* semantics (ARM 9.8).

deadline handler
budget handler

Attribute:	**T'Deadline_Handler, T'Budget_Handler**
Aspect:	Deadline_Handler, Budget_Handler
Type:	Ada.Cyclic_Tasks.Response_Handler
Type Definition	**access protected procedure** (T : Ada.Task_Indentification.Task_Id);
Default:	**null**
Valid:	T'Deadline_Response = Handler, T'Budget_Response = Handler

The protected procedure called by the run-time if the corresponding _Response attribute is set to Handler. If a task has a _Response aspect set to Handler then the corresponding _Handler shall be set.

The handler is passed the Task_Id of the task causing the timing violation to allow a handler to be shared amongst a set of tasks.

execution server

Attribute:	**T'Execution_Server**
Aspect:	Execution_Server
Type:	Ada.Execution_Servers.Execution_Server
Type Definition	Execution_Server (<>) **is abstract tagged limited private;**
Default:	Ada.Execution_Server.No_Server
Valid:	any task type

Specifies the execution server the task will be attached to after activation.

Task Semantics

Without the cyclic section, the semantics of a task remains unchanged from Ada 2012. With the cyclic section, the semantics of the task becomes:

- The sequential_sequence_of_statements executes after the activation of the task in accordance with the selected task dispatching policy. The sequential section allows the user to initialise the task before the cyclic execution begins.
- A task body with a cyclic_sequence_of_statements shall have a non-False Cyclic_Task aspect applied to the task type.
- Under normal operation, the task will not terminate.
- A task body's exception_handlers shall handle exceptions raised from the sequential_sequence_of_statements and sequential_sequence_of_statements. The task will terminate after the handler completes.

- The execution of a cyclic_sequence_of_statements is called a cycle.
- A task shall only execute a single cycle at any one time.
- A cycle is commenced when a cycle event is raised.
- Periodic tasks have their cycle events generated by the run-time. The first cycle event occurs at the time resulting from the sum of Cyclic_Execution_Start_Time and the task's Cycle_Phase. Subsequent cycle events shall be raised by the run-time at T'Cycle_Period intervals from the first event. Cycle events are ignored while an existing cycle executes.
- Aperiodic and sporadic tasks have cycle events raised by the new_cycle_statement. If the target task is still executing a cycle when new_cycle_statement is called, the run-time will react in accordance with the target task's Early_Event_Response attribute.
- The Cycle_MIT defines a sporadic task's minimum inter-arrival time for cycle events. If a cycle event raised by new_cycle_statement occurs within Cycle_MIT of the last cycle event, the run-time reacts in accordance with the target task's Early_Event_Response attribute.
- A new_cycle_statement with a task object which is not sporadic or aperiodic shall raise Tasking_Error at run-time. A new_cycle_statement executed before the sum of Cyclic_Execution_Start_Time and the target task's Cycle_Phase shall raise Tasking_Error.
- Yielding tasks create their own cycle events when completing a cycle. The first cycle event is raised by the run-time at the sum of Cyclic_Execution_Start_Time and the task's Cycle_Phase.
- Once a task completes a cycle it shall yield the processor. If there is no pending cycle event the task is blocked from executing until the next cycle event.
- The absolute deadline for a cycle is the time of the cycle event plus the task's Relative_Deadline attribute. If a cycle does not complete before its deadline the run-time shall react in accordance with the task's Deadline_Response.
- A task commences each cycle with the Remaining_Execution_Budget attribute set to the value of the task's Execution_Budget attribute.
- If the cycle consumes all of its execution budget before completing, the run-time shall react in accordance with the task's Budget_Response.
- For periodic and sporadic tasks the Execution_Budget and Relative_Deadline for a task shall be no greater than the task's Cycle_Period and Cycle_MIT respectively.
- If the task has an Execution_Server attribute specified, the task will only execute when allowed so by the server. If specified via an aspect, the task will be added to the execution server after activation. The task will activate at the higher of the task's or the activator's Priority attributes.

- For the Dynamics package: changes to Priority, Absolute_Deadline, Remaining_Execution_Budget and the _Response attributes take effect immediately. Changes to Cycle_MIT, Relative_Deadline and Execution_Budget attributes take effect on the next cycle. Changes to Cycle_Period take effect on the second subsequent cycle.[2]

Execution Servers

Execution servers allow a set of tasks to share periodic access to a processor, enabling the scheduling of aperiodic and sequential tasks amongst other cyclic tasks while preserving the overall periodicity of the system: making offline schedulability analysis possible. Due to the variety of different execution server implementations (Buttazzo, 2011), the *Cyclic Task Specification* does not prescribe a specific implementation. Instead, a library provides a general execution server interface to facilitate the creation and termination of execution servers, and to add and remove tasks from these servers. Run-times and users use this interface to provide their execution server implementations without requiring additional compiler support outside the new Execution_Server task aspect.

library packages

The Ada.Execution_Servers package provides the general interface (Appendix A). Implementations derive from the abstract tagged type Execution_Server:

```
type Execution_Server (<>) is abstract tagged limited private;
```

The Execution_Server object need not directly implement the execution server; instead, it may act as a proxy to other run-time objects. The type uses an unknown discriminant to force an Execution_Server object's initialisation — and, if needed, registration of the execution server with the run-time — at the time of the object's creation through the abstract function New_Execution_Server:

```
function New_Execution_Server
  (Priority          : System.Any_Priority;
   Period            : Time_Span;
   Phase             : Time_Span := Real_Time.Time_Span_Zero;
   Execution_Budget  : Time_Span := Real_Time.Time_Span_Last;
   Relative_Deadline : Time_Span := Real_Time.Time_Span_Last;
   CPU               : CPU_Range := Not_A_Specific_CPU)
  return Execution_Server is abstract;
```

The Execution_Server object requires initialisation with its parameters at the time of object creation since the run-time can add tasks to the object before the end of the declarative part the object is a member.

At minimum, each execution server requires a priority, period, phase and CPU assignment to provide the periodic attributes of the server. Each server also has an execution budget and relative deadline property, the latter permitting the server to operate under any of the existing task dispatching policies. The properties are specifiable as part of the initialisation procedure with functions available in the Ada.Execution_Servers package to read these core server parameters at run-time. It is recommended to use New_Execution_Server to initialise new Execution_Server objects to simplify the identification of server parameters, for both users and tools operating on the software.

2. Simplifying the case when the task is in-between cycles and the cycle period shortens.

The Ada.Execution_Servers package only offers environment defined execution servers with static properties. Tasks are assigned to these servers via their Execution_Server aspect and no facility exists to remove member tasks. Consequently, these servers suit schedulability analysis by ensuring timing properties and members do not change at run-time.

Two child packages support greater run-time and user flexibility for non-hard real-time systems: Ada.Execution_Servers.Dynamics and Ada.Execution_Servers.User_Servers (Appendix A). The Dynamics package extends the Execution_Server type with the Dynamic_Execution_Server type, providing standard procedures to dynamically change execution server properties at run-time in addition to adding and removing member tasks.

Similarly, the User_Servers package offers the same flexibility as the Dynamics package — with User_Execution_Server extending from Dynamic_Execution_Server — but enables users to implement their own execution servers. This distinction provides a visible and semantic separation of environment and user supplied execution servers, with the run-time required to call the user's Add_Task_To_Server procedure when adding a task specified via the task's Execution_Server aspect. The run-time does not need do this for other execution servers, offering flexibility in their implementation.

model

Conceptually, the *Specification* models execution servers like periodic tasks, with execution servers consisting of cycles which commence at periodic intervals as governed by the server's period and phase properties. These cycles, called *execution windows*, provide a window for tasks assigned to the server to execute. Consequently, instead of directly executing like a periodic task, execution servers dispatch member tasks in their place, with tasks dispatched from a first-in, first-out queue.

When a member task temporarily inherits a priority because of intertask communication (ARM D.1), the task leaves the execution server and adopts the inherited priority. While outside the execution server, the task does not consume the server's execution budget nor shall the task block when the execution window closes until its base priority restores.

implementation choices

Execution server implementations have considerable flexibility to choose how they behave. The first decision an implementation needs to consider is when an execution window closes. Two primary methods exist:

- The execution window closes once an absolute deadline passes, with the deadline calculated using the server's relative deadline property.
- The execution window closes when the execution budget of the cycle is exhausted. The server's execution budget property sets the budget of the cycle.

An execution server may choose to employ both techniques to constrain how long a window remains open for if its budget is not consumed within an allotted time. Alternatively, an execution server may choose another method to determine when to close an execution window.

Further, execution servers have the prerogative to choose how member and non-member tasks consume their execution budgets, and what happens when an execution window is open but the server has no tasks to dispatch. For the latter, the server may let lower priority tasks execute inside its window or may block the processor until it can dispatch a member task.

Ravenscar Profile Support

For Ravenscar Profile systems, most of the cyclic task behaviour is available. Removed are the Ada.Cyclic_Tasks.Dynamics, Ada.Execution_Servers.Dynamics and Ada.Execution_Servers.User_Servers packages, as dynamically changing cyclic task properties introduces non-deterministic tasking behaviour. Also removed are the deadline and execution budget responses Abort_Cycle and Abort_Task as they introduce asynchronous transfer of control and task termination.

Examples

A number of simple examples demonstrate the power and flexibility of the *Cyclic Task Specification*.

periodic counter

A simple periodic task counts the number of elapsed seconds, printing the value of the counter each cycle. Using the new syntax:

Listing 25
A simple period counter using the *Cyclic Task Specification*.

```
task Counter
  with Priority     => 15,
       Cyclic_Task  => Periodic,
       Cycle_Period => Seconds (1);

task body Counter is
   J : Natural := 0;
begin
   Put_Line ("Counter Start");
   Put (J);
cycle
   J := J + 1;
   Put (J);
end Counter;
```

Notice how the body of the task does not contain any task timing information; instead, that information can be found in a clear and concise format on the task specification. The only cyclic information the user needs to care about in the body of the task is their cyclic code and the initialisation of the task.

A user though may desire a collection of counter tasks that increment at different rates. Reusing the same body, the task declaration would become:

```
task type Counter (Rate_In_Seconds : Positive)
  with Priority     => 15,
       Cyclic_Task  => Periodic,
       Cycle_Period => Seconds (Rate_In_Seconds);
```

The user can now create counters with different increment rates:

```
Counter_1 : Counter (Rate_In_Seconds => 1);
Counter_2 : Counter (Rate_In_Seconds => 5);
```

Or even an array of counters:

```
Array_Of_Counters : array (1 .. 100) of
                      Counter (Rate_In_Seconds (60));
```

sporadic red button monitor

In this example, a sporadic task counts the number of times a big red button is pressed, ignoring any subsequent presses for five second. Once the button has been pressed more than ten times the system will self-destruct. Note this example demonstrates the use of an exception handler and how the task abstraction provides a scope to make the Button_Press_Counts type visible only to the body of the task.

Listing 26
A sporadic task demonstrated by a red button monitor task.

```
task Red_Button_Monitor
  with Priority              => 15,
       Cyclic_Task           => Sporadic,
       Cycle_MIT             => Seconds (5),
       Early_Event_Response  => Ignore;

task body Red_Button_Monitor is
   type Button_Press_Counts is range 0 .. 10;
   Press_Count : Button_Press_Counts := 0;
begin
   null;
cycle
   Press_Count := Press_Count + 1;
exception
   when Constraint_Error =>
      Self_Destruct;
end Red_Button_Monitor;
```

An interrupt handler attached to the button would inform the task the button has been pressed through the new_cycle_stastement:

Listing 27
An example interrupt handler that will cause the red button monitor to run when the button is pressed.

```
protected Button is
   procedure Button_Handler with Attach_Handler => Button_IRQ;
end Button;

protected body Button is
   procedure Button_Handler is
   begin
      Clear_Button_Interrupt;
      new cycle Red_Button_Monitor;
   end Button_Handler;
end Button;
```

sporadic state communicator

In this example, a sporadic task transmits the state of a protected object when requested to. Successive transmissions are separated by one second, with any early request postponed until this time elapses. Communication has to completed within 500 milliseconds otherwise the communication is abandoned. The task has an execution budget of 100 milliseconds, with any cycle exceeding the budget logged.

Listing 28
Sporadic state communicator task demonstrating deadline and budget enforcement.

```
protected Handler_Object is
   procedure Exhaustion (T : Task_Id);
end Handler_Object;

protected body Handler_Object is
   procedure Exhaustion (T : Task_Id) is
   begin
      Log_Budget_Exhaustion (T);
   end Exhaustion;
end Handler_Object;

task State_Communicator
  with Priority            => 15,
       Cyclic_Task         => Sporadic,
       Cycle_Phase         => Milliseconds (200),
       Cycle_MIT           => Seconds (1),
       Early_Event_Response => Delay;
       Relative_Deadline   => Milliseconds (500),
       Deadline_Response   => Abort_Cycle,
       Execution_Budget    => Milliseconds (100),
       Execution_Response  => Handler,
       Execution_Handler   => Handler_Object.Exhaustion'Access;

task body State_Communicator is
   Active_Comm   : Boolean := False;
   State_To_Send : PO_State;
begin
   Comm_Device.Set_Up;

cycles
   if Active_Comm then
      Comm_Device.Reset_Channel;
   end if;

   State_To_Send := The_Protected_Object.Get_State;
   Comm_Device.Send (State_To_Send);
end Cyclic_Task;
```

This example demonstrates how the *Cyclic Task Specification's* cyclic task abstraction makes it easy to implement complex, but common, cyclic task behaviours. The user only needs to describe the task's cyclic behaviour on the task type and provide the necessary handlers, with the Ada environment taking care of the rest. Additionally, this task can be used with any of Ada's task dispatching policies without modification. Finally, notice a user can provide others the task's specification and they will receive the cyclic attributes of the task; allowing others to gather the task's timing requirements and its impact on the system without the task's body or relying on possibly inconsistent documentation.

aperiodic counter

This final example revisits the *Periodic Counter* by changing its behaviour to be aperiodic, modifying it to counting the number of times the counter has been invoked. Additionally, the aperiodic task is attached to a Priority Server, an execution server implementation.

Listing 29
An aperiodic counter demonstrating aperiodic tasks attached to execution servers..

```
My_Priority_Server : Priority_Server :=
  New_Execution_Server (Priority          => 16,
                        Cycle_Period      => Seconds (10),
                        Cycle_Phase       => Seconds (1),
                        Execution_Budget  => Seconds (3),
                        Relative_Deadline => Seconds (4));

task Aperiodic_Counter
  with Cyclic_Task      => Aperiodic,
       Execution_Server => My_Priority_Server;

task body Aperiodic_Counter is
   J : Natural := 0;
begin
   Put_Line ("Counter Start");
   Put (J);
cycle
   J := J + 1;
   Put (J);
end Aperiodic_Counter;
```

With the counter invoked with:

```
new cycle Counter;
```

Notice the body remains unmodified between the *Periodic Counter* and *Aperiodic Counters*: changes only occur to the task's specification. This demonstrates the power of the *Specification's* cyclic task abstraction: removing all cyclic behaviour statements out of body.[3] As a result, the cyclic task abstraction makes cyclic tasks considerably more easier to read and write than existing techniques, with the body consisting of only the statements and variables that make the task unique. Additionally, the tasks do not require changes to operate on different task dispatching policies, boosting code reuse.

Implementation

Implementing the *Cyclic Task Specification* comes into two parts: expansion of the syntax and provisioning the necessary run-time support. The run-time support required by the *Specification* depends on the existing tasking services provided by run-time — and its underlying operating system if applicable — and the chosen expansion form for the new tasking syntax.

The simplest part of the cyclic task syntax to incorporate into an existing Ada

3. Unless changes to the task attributes are required at run-time: then the body will contain library calls to the appropriate Ada.Cyclic_Tasks.Dynamics procedures.

compiler is the new task attributes and aspects. These attributes and associated aspects follow the same form as the existing Priority and CPU task attributes, making them trivial to incorporate by following these existing implementations. An exception lies with the Cyclic_Task aspect, which also requires the compiler to check the appropriateness of the task's body cyclic section.

Two different approaches exist for expanding cyclic task bodies. Since a task's cyclic kind is static, one approach sees a task body expanded into the corresponding direct cyclic task implementation outlined in *Implementing Cyclic Tasks in Ada* on page 16. This tactic favours compilers providing only simple cyclic tasks without deadlines and execution budgets. As Chapter 2 showed, these forms of cyclic tasks require little extra code to create the required cyclic task semantics from a sequential task.

A more balanced approach better supporting the *Specification* leverages the essence of the cyclic task model: a cycle commences in response to a cycle event once a previous cycle has concluded. Here, the burden of implementation falls onto the run-time: controlling when the task can commence a new cycle. Consequently, the expansion of a cyclic task can transform a task body's task_sequence_of_statement:

```
sequential_sequence_of_statements
[cycle
  cyclic_sequence_of_statements]
[exception
  exception_handler
 {exception_handler}]
```

into a handled_sequeunce_of_statements which the compiler can already handle:

```
sequence_of_statements
[loop
   Wait_For_Cycle_Event;
   sequence_of_statements
 end loop]
[exception
   exception_handler
  {exception_handler}]
```

The run-time provides the Wait_For_Cycle_Event procedure, blocking the task until the next cycle can commence. The procedure also signals to the run-time when the task completes a cycle, and manages cycle deadline and execution budget enforcement. The management of cycle events — the generation of periodic events and enforcement of MIT — can occur here or alternatively handled by the underlying operating system or kernel. How the run-time provides these services is dependent on the existing services offered by run-time or its underlying system. Chapter 5 describes one such cyclic task run-time implementation for the real-time executive Acton.

For execution servers, the *Specification* requires run-times to determine themselves how to provide the appropriate representation within the execution server model. Run-times can choose to internally implement execution servers as, for example:

- like tasks, but dispatching their member tasks whenever the run-time selects the server for dispatch;
- using Ada 2012's group execution time budgets (ARM D.14.2);
- using timers and asynchronous task control to resume and suspend tasks as the server's execution window opens and closes; or
- using operating system provided service like POSIX's sporadic servers.

Note a task-like representation within the run-time enables execution server implementations to be portable across different task dispatching policies.

Ada Semantic Interface Specification

The Ada Semantic Interface Specification (ASIS) standard (ISO/IEC, 1999) provides a standardised library interface enabling access to the semantic and syntactic information of an Ada program (AdaCore, 2014). Its existence facilitates the development of code analysis tools independently from the underlying Ada environment and lowers the entry barrier for new code analysis tools. The interface consists of queries tools can perform on an Ada program's abstract syntax tree.

Modifying the task body syntax and the introduction of the new_cycle_statement to support the cyclic task abstraction inside Ada necessitates changes to ASIS so tools can support and take advantage of the new syntax. The changes documented here affect the ASIS, ASIS.Declarations and ASIS.Statements packages.

ASIS

Within the ASIS package, the enumeration type Statement_Kinds (ASIS 3.9.20) expands to include a new enumeration literal A_New_Cycle_Statement, allowing identification of the new_cycle_statement:

Listing 30 Amendment to ASIS.Statement_Kinds (asis.ads).

```
-------------------------------------------------------------------
--  3.9.20   type Statement_Kinds
-------------------------------------------------------------------
--  Statement_Kinds - classifications of Ada statements
--   Literals                            -- Reference Manual
-------------------------------------------------------------------

   type Statement_Kinds is (..., A_New_Cycle_Statement);
```

ASIS.Declarations

The ASIS.Declarations package provides the Body_Statement query to return the list of statements and pragmas for a given body declaration (ASIS 15.22). Body declarations are defined to include subprograms bodies, entry bodies, package bodies and, most relevantly, task bodies. While the *Cyclic Task Specification* now defines two distinct sets of task body statements, for backwards capability the Body_Statement query on task body declarations remains. The query will return a single set of statements (appending the cyclic statements

to the end of the sequential statements) but such a query is obsolescent and only for compilers not implementing cyclic tasks or supporting legacy tools:

```
-----------------------------------------------------------------
--  15.22 function Body_Statements
-----------------------------------------------------------------

   function Body_Statements
     (Declaration     : Asis.Declaration;
      Include_Pragmas : Boolean := False)
      return            Asis.Statement_List;

-----------------------------------------------------------------
--  Declaration     - Specifies the body declaration to query
--  Include_Pragmas - Specifies whether pragmas are to be
--                    returned
--
--  Returns a list of the statements and pragmas for the body, in
--  their order of appearance.
--
--  Returns a Nil_Element_List if there are no statements or
--  pragmas.
--
--  Appropriate Declaration_Kinds:
--       A_Function_Body_Declaration
--       A_Procedure_Body_Declaration
--       A_Package_Body_Declaration
--       A_Task_Body_Declaration   - obsolescent, not recommended
--       An_Entry_Body_Declaration
--
--  Returns Element_Kinds:
--       A_Pragma
--       A_Statement
--
--  --|AN Application Note:
--  --|AN
--  --|AN This function is an obsolete feature when provided an
--  --|AN A_Task_Body_Declaration. Use Sequential_Body_Statements
--  --|AN and Cyclic_Body_Statements instead.
```

Listing 31 Amendment to ASIS.Declarations.Body_Statement interface (asis-declarations.ads).

Replacing the Body_Statement query is the new Sequential_Body_Statements and Cyclic_Body_Statements queries, returning the list of statements and pragmas for the task's sequential and cyclic sections respectively (Listing 32 on page 58). Tasks without a cyclic section will return Nil_Element_List in response to the Cyclic_Body_Statements query.

ASIS.Statements

The final change to ASIS sees the introduction of a new statement query: New_Cycle_Tasks, allowing an ASIS tool gather the tasks referred to in a new_cycle_statement (Listing 33 on page 59).

Listing 32 Additions to ASIS.Declarations.interface (asis-declarations.ads).

```
-----------------------------------------------------------------
--  15.?? function Sequential_Body_Statements
-----------------------------------------------------------------

    function Sequential_Body_Statements
      (Declaration     : Asis.Declaration;
       Include_Pragmas : Boolean := False)
       return            Asis.Statement_List;
-----------------------------------------------------------------
--  Declaration     - Specifies the body declaration to query
--  Include_Pragmas - Specifies whether pragmas are to be
--                    returned
--
--  Returns a list of the statements and pragmas for the
--  sequential body section of a task, in their order of
--  appearance.
--
--  Returns a Nil_Element_List if there are no statements or
--  pragmas.
--
--  Appropriate Declaration_Kinds:
--       A_Task_Body_Declaration
--
--  Returns Element_Kinds:
--       A_Pragma
--       A_Statement


-----------------------------------------------------------------
--  15.?? function Cyclic_Body_Statements
-----------------------------------------------------------------

    function Cyclic_Body_Statements
      (Declaration     : Asis.Declaration;
       Include_Pragmas : Boolean := False)
       return            Asis.Statement_List;

-----------------------------------------------------------------
--  Declaration     - Specifies the body declaration to query
--  Include_Pragmas - Specifies whether pragmas are to be
--                    returned
--
--  Returns a list of the statements and pragmas for the
--  cyclic body section of a task, in their order of appearance.
--
--  Returns a Nil_Element_List if there are no statements or
--  pragmas.
--
--  Appropriate Declaration_Kinds:
--       A_Task_Body_Declaration
--
--  Returns Element_Kinds:
--       A_Pragma
--       A_Statement
```

```
-----------------------------------------------------------------
--  18.?? function New_Cycle_Tasks
-----------------------------------------------------------------

   function New_Cycle_Tasks
     (Statement : Asis.Statement)
      return      Asis.Expression_List;

-----------------------------------------------------------------
--  Statement    - Specifies the new cycle statement to query
--
--  Returns a list of the task names from the NEW CYCLE statement,
--  in their order of appearance.
--
--  Appropriate Statement_Kinds:
--       A_New_Cycle_Statement
--
--  Returns Element_Kinds:
--       An_Expression
```

Listing 33 Additions to ASIS.Statements.interface (asis-statements.ads).

ASIS Tools and Cyclic Tasks

The *Cyclic Task Specification's* initial benefit to users is easy to author cyclic tasks that are simple to understand: achieved by reducing the amount of code required to express a cyclic task and presenting it in a standardised approach. With the changes to ASIS, the benefits also apply to Ada code analysis tools.

Code analysis tools can now extract cyclic task information right from a program's code because of the standardised attributes. The utility of the *Specification* increases when using static cyclic attributes, because they appear as aspects on the definition of task types. While more challenging, analysis of tasks with dynamic attributes is still possible due to the standardised library procedures that change the cyclic attributes.

Since code analysis tools can now extract cyclic task properties direct from Ada programs, a new class of tools can now help ease the development of reliable and maintainable software. For example, current schedulability analysis tools require users to either manually enter cyclic task attributes or provide them through specially annotated code. Now a schedulability analysis tool can examine an Ada program directly and perform its analysis using the task attributes the analyser pulled from the program's code; all without user input. In this scenario, the user has greater confidence in the analysis because the analyser *will be using the exact same task attribute values as the program will at run-time*. Mistakes introduced during the transfer of attributes between code and analyser can no long occur because the analyser has access to the same information the compiler uses.

Similarly, WCET tools can now identify the cyclic section of a task themselves. When combined with the programs object code, analysis becomes more automated because the user no longer has to explicitly tell the tool where the cyclic task sections are situated. However, the real benefit for WCET tools lies in the ability for them to directly write their results back into the program's code in the form of execution budgets and *have them enforced at run-time.*

As the two examples show, enabling tools to directly read and write cyclic task attributes facilitates greater reliability and maintainability because tools can now access the same semantic data as the compiler. Furthermore, the tools become more autonomous as they can find more of the information they require in a standardised form. Consequently, identification of code defects can occur much quicker and earlier because the seamless integration of the code analysis tools into the Ada compiler's toolchain becomes possible. For real-time systems, this opens up greater reliability: timing defects become more easily discovered with less involvement from the user since the toolchain can now integrate schedulability and WCET analysis.

Demonstrating the ability for code analysis tools to pull cyclic task attributes from a program is a simple ASIS-based tool call Acton Analyser: built to aid the dissertation's new real-time executive, Acton (Chapter 5). Since Acton requires the pre-allocation of certain kernel resources at the executive's compilation time, the tool plays an important role helping users configure Acton so sufficient resources are available. The tool tables the tasks and protected objects defined within an Acton-based program along with their attributes and Acton's configuration. Additionally, the tool documents the program's stack requirements to ensure the program can fit within the target microcontroller's RAM. Running the Acton Analyser on Frank — the LEGO robot from Chapter 2 — produces the following output for Frank's task units:

```
> acton_analyser -tu frank_the_robot
=================================================================
TASK UNITS
-----------------------------------------------------------------
Task Identifier      : Frank.Datalink.Datalink
Declaration Location : src/frank-datalink.ads:9
Body Location        : src/frank-datalink.adb:9
Cyclic Task          : PERIODIC
Task Priority        : 10
Task Storage Size    : 1024 bytes
Cycle Period         : 100 milliseconds
Execution Budget     : 40 milliseconds
Relative Deadline    : 100 milliseconds

Main Task            : Frank_The_Robot
Declaration Location :
Body Location        : src/frank_the_robot.adb:0
Cyclic Task          : NO
Task Priority        : 14
Task Storage Size    : 4096 bytes

Task Identifier      : Frank.Position.Position_Tracker
Declaration Location : src/frank-position.ads:29
Body Location        : src/frank-position.adb:8
Cyclic Task          : PERIODIC
Task Priority        : 14
Task Storage Size    : 1024 bytes
Cycle Period         : 4 milliseconds
Execution Budget     : 10 milliseconds
Relative Deadline    : 30 milliseconds
```

```
Task Identifier      : Mindstorms.NXT.I2C.I2C_Controller
Declaration Location : src/mindstorms-nxt-i2c.ads:14
Body Location        : src/mindstorms-nxt-i2c.adb:90
Cyclic Task          : APERIODIC
Task Priority        : 25
Task Storage Size    : 1024 bytes
Cycle Phase          : 10 milliseconds
Execution Budget     : 1 milliseconds
Relative Deadline    : 5 milliseconds

Task Identifier      : Mindstorms.NXT.AVR.AVR_Communicator
Declaration Location : src/mindstorms-nxt-avr.ads:164
Body Location        : src/mindstorms-nxt-avr.adb:136
Cyclic Task          : PERIODIC
Task Priority        : 28
Task Storage Size    : 1024 bytes
Cycle Period         : 2 milliseconds
Cycle Phase          : 20 milliseconds
Execution Budget     : 1 milliseconds
Relative Deadline    : 1 milliseconds

Task Identifier      : Frank.Obstacles.Gather_Obstacle_Distance
Declaration Location : src/frank-obstacles.ads:19
Body Location        : src/frank-obstacles.adb:20
Cyclic Task          : PERIODIC
Task Priority        : 14
Task Storage Size    : 2048 bytes
Cycle Period         : 50 milliseconds
Cycle Phase          : 1 seconds
Execution Budget     : 2 milliseconds
Relative Deadline    : 10 milliseconds

Task Identifier      : Frank.Control.Frank_Control_Logic
Declaration Location : src/frank-control.ads: 37
Body Location        : src/frank-control.adb: 42
Cyclic Task          : PERIODIC
Task Priority        : 14
Task Storage Size    : 2048 bytes
Cycle Period         : 100 milliseconds
Execution Budget     : 5 milliseconds
Relative Deadline    : 10 milliseconds

Task Identifier       : Frank.Datalink.Manager.Datalink_Agent
Declaration Location  : src/frank-datalink-manager.ads: 25
Body Location         : src/frank-datalink-manager.adb: 49
Cyclic Task           : APERIODIC
Task Priority         : 1
Task Storage Size     : 2048 bytes
Execution Budget      : 2 milliseconds
Relative Deadline     : 10 milliseconds
------------------------------------------------------------------
* Total number of task units :  8
=================================================================
```

Language Impact and Flexibility

Providing cyclic tasks through a library-based approach is preferred over incorporating a cyclic task abstraction in an existing language's syntax because the latter forces changes to the language's compilers and run-times, and potentially existing programs. Consequently, significant change to the language quickly diminishes any appetite for language modification, more so when the changes affect the wider Ada community and does not provide the flexibility demanded by its target users.

While the cyclic task abstraction presented by the *Cyclic Task Specification* follows an approach analogous to Baker and Lucas, it does so without significant complexity: not requiring a new task class needing to be treated differently by other language features. For example, a task with cyclic semantics behaves no differently in rendezvous than a task without. In detail, the cyclic semantic extension to the task body consists of three parts:

- a sequence of code, executed cyclically;
- a cycle release mechanism which releases cycles in response to cycle events; and
- a mechanism to enforce cycle deadlines and execution budgets.

By containing the cyclic code in its own section, not nested inside a sequential sequence of statements, and using an implied release mechanism, the syntax of the cyclic section does not allow statements (including intertask communication) to span across task cycles. Under normal conditions, when a task does not miss its deadline or exhaust its execution budget, all cyclic statements must complete before the cycle can. Thus, a task executing a cycle interacts no differently with other tasking features than an existing sequential task because the cycle fully contains the interaction.

If a task does miss its deadline or exceeds its execution budget, support for the three run-time responses (call handler, abort cycle and abort task) already exist in Ada. Handlers behave in the same manner as handlers for timing events and execution time timers (ARM D.15 and D.14.1), while the abort responses follow Ada's abort of a sequence of statements and task abortion respectively (ARM 9.8).

Thus, the cyclic task abstraction approach taken by the dissertation confines changes to the language to the task unit alone. The design also ensures Ada's existing tasking semantics remains unmodified when a cyclic section is not present. However, ramifications to the language remain through:

- increase complexity of Ada tasks; and
- a new keyword.

As the chapter has shown, the *Cyclic Task Specification* is extensive and accordingly adds complexity to Ada. Reducing the *Specification's* impact however, only the cyclic task syntax and the core of the cyclic task model needs implementing within the core language. The core model retains the concept of a cyclic task as an execution unit executing a set of cycles sequentially in response to cycle

events; removed from the model is all concept of time. Consequently, the cyclic task subset for the core language consists of:

- Task body syntax
- New cycle statement
- Cyclic_Task attribute
- Yielding task semantics
- Aperiodic task semantics

The core cyclic task subset provides basic cyclic task support for not only real-time systems, but also for other system types. For example, server software typically implements a cyclic task model: receiving a request, processing it, and returning a response before repeating the processes with another request. Such a server task can be implemented using a yielding task with the receive request statement as the first statement of the cycle. Aperiodic tasks can also be valued in event-driven systems where a task needs to run in response to an event. Consequently, the core cyclic task model serves the needs for a wide range of users, using a model simple and flexible enough to be described in a single sentence.

The rest of the *Specification* (periodic and sporadic task semantics; their respective attributes; deadline and execution time enforcement; and execution server support) sit within the *Real-Time Systems Annex*. Doing so demonstrates the flexibility of the core cyclic task model to support the specialised needs of a subset of Ada users, in this case real-time users, through semantic extension.

Furthermore, the ability to extend the core cyclic task model addresses the concern held within the Ada community that high-level abstractions are too restrictive and frequently do not provide the functionality users expect from the abstractions. The fear justifiably grew out of early Ada experience, where abstractions went unused or — worse — impeded the development of reliable and maintainable systems in a cost effective manner (for example real-time systems and Ada 83's tasks). In many ways, an abstraction's usability and constraints it places on users is more important than the complexity the abstraction adds to the language or its implementability.

The *Cyclic Task Specification* avoids this problem by building on a core cyclic task model. Fundamentally, this is what differentiates the *Specification* from Baker's and Lucas' periodic task abstractions: while usefully providing periodic tasks, the abstractions provided nothing for users wanting aperiodic and sporadic tasks, or requiring deadline and execution budget enforcement. In these cases, a user will end up not using these periodic task abstractions because they apply only to a specific use case and have nothing in common with the other tasks the user will implement.

By providing a core cyclic task model, the *Specification* can conceivably support any cyclic task model on top of it. With the dissertation's focus on real-time systems, the *Specification* integrates support for real-time tasks within its cyclic task model; aiming to support common forms of real-time tasks as described in real-time systems text-books (Burns & Wellings, 2009; Buttazzo, 2011; Ko-

petz, 2011). Support for real-time cyclic tasks builds upon the core cyclic task model through additional run-time semantics and new task attributes that guide the semantics.

The *Specification* can easily expand to incorporate new constraints on cycles or introduce new forms of cyclic tasks by implementing the necessary run-time support and providing the associated new task attributes. For example, HAL/S provides the ability to activate or deactivate periodic tasks based on a time or event. An extension to the *Specification* could deliver similar functionality by adding appropriate attributes to describe the conditions for the activation and deactivation, and then implementing the desired behaviour in the run-time.

Alternatively, an Ada environment may wish to provide a cyclic task that maintains a certain processor bandwidth over a defined period, with the run-time raising cycle events when the task is below its bandwidth and not raising cycle events when the task is over it. To implement, the Ada environment would provide the required run-time and compiler support. This would include extending the Ada.Cyclic_Tasks.Task_Kind type to include a new implementation-defined enumeration item Bandwidth and define new implementation-defined attributes T'Bandwidth and T'Bandwidth_Period to drive the run-time's generation of cycle events. Underpinning this ability is the Specifications sole use of attributes and aspects to modify the behaviour of cyclic tasks and Ada's support for implementation defined attributes (ARM 4.1.4) and aspects (ARM 13.1.1).

These examples highlight the weakness of the *Cyclic Task Specification's* strength: relying on the run-time to control the release and enforcement of each cycle. While this approach hides the implementation from the user, the user can only take advantage of the cyclic task model if their Ada environment provides the behaviour the user is after. But this is no different from the approaches other languages, from PEARL to RTSJ. The strength of Ada and the *Specification* lies with the ability to use the advantages of the Specification where possible, but also implement less used cyclic task behaviour using the existing sequential task abstraction provided by Ada. Unlike other languages, the language can incorporate new cyclic behaviour without impacting existing users.

Finally, the largest impact the *Cyclic Task Specification* has on Ada is the new keyword **cycle**. A new keyword causes compatibility problems with existing software since a valid name becomes invalid, requiring software to change all references to the word before using the updated language. The use of the word **cycle** as a keyword is particular problematic as the ARM uses Cycle as a parameter to a number of language-defined mathematical functions (ARM A.5.1, G.1.1, G.3.2). Consequently, Ada will need to rename the Cycle parameter or alternatively choose a different keyword to support the *Specification*.

Two arguments exist for choosing **cycle** as the keyword. It works well as a verb as part of the task body syntax — **task body** ... **is** ... **begin** ... **cycle** ... **end** — to denote the cyclic section of the task. Additionally, it reads well as the noun in the new_cycle_statement — **new cycle** Task_1, Task_2; — in a similar way to the **abort** statement.

If the impact of changing the ARM's existing use of Cycle is too high, a recommended alternative keyword is **cycles**. While not as elegant, **cycles** still works as a verb for the task body syntax — **task body** ... **is** ... **begin** ... **cycles** ... **end**. However, it does not work for the new_cycle_statement since the state-

ment only releases one cycle per task. Consequently, if choosing **cycles** as the keyword, the recommendation is to replace the new_cycle_statement with a new non-modifiable cycle attribute: T'New_Cycle. Querying the attribute will cause the run-time to raise a cycle event for the prefixed task using the same semantics as the new_cycle_statement. The downside of this approach is the loss of new_cycle_statement ability to raise cycle events in a list of tasks within a single statement and the abuse of attributes to raise cycle events.[4]

Comparison with Existing Ada Approaches

Both the current chapter and Chapter 2 have documented the advantages and flaws of the different approaches for implementing cyclic tasks. Here the analysis broadens to a direct comparison of the *Cyclic Task Specification* against the two flexible approaches to implementing cyclic tasks under Ada 2012: the direct Ada implementation and the *Real-Time Utilities Library*. The generic library approach provided by GNAT-Coll was not considered since the approach does not cover the dissertation's cyclic task model.

The comparison uses two cyclic task examples presented in this chapter: the *Periodic Counter* from Listing 25 on page 51 and the *Sporadic State Communicator* from Listing 28 on page 53. These examples provide both a simple and a complex cyclic task from which comparison of the three different approaches can be made. Boxes 1 and 2 on pages 60–65 present the different implementation approaches for the two examples respectively.

In addition to the results from the comparisons, the *Cyclic Task Specification* provides a number of additional advantages:

- Cyclic tasks are independent of the underlying task dispatching policy. Thus, a task with a deadline can run unmodified on any Ada task dispatching policy and even moved between policies run-time. Other approaches do not provide this facility because the Ada's EDF task dispatching policy requires the use of a different delay procedure.
- ASIS tools can read and write the static attributes of a cyclic task, enabling better automation from real-time tools like schedulability and WCET analysers.

4. Since an attribute reference is meant to be a query on a particular characteristic of the entity (ARM 4.1.4). The alternative — a library procedure Ada.Cyclic_Tasks.New_Cycle (T : Ada.Task_Identification.Task_Id) — is less ideal since users will still need an attribute reference and include the Ada.Cyclic_Tasks package: Ada.Cyclic_Tasks.New_Cycle (Task_Name'Identity).

Periodic Counter Comparison

Using the Cyclic Task Specification

Listing 34
The periodic counter using the *Cyclic Task Specification*.

```
task Counter
  with Priority     => 15,
       Cyclic_Task  => Periodic,
       Cycle_Period => Seconds (1);

task body Counter is
   type Private_Natural is new Natural;
   J : Private_Natural := 0;
begin
   Put_Line ("Counter Start");
   Put (J);
cycle
   J := J + 1;
   Put (J);
end Counter;
```

- `with Priority => 15,` — initial attributes appear on task definition; forced static with the corresponding pragma Restrictions
- `Cyclic_Task => Periodic,` — no need for body to determine static attributes
- `Cycle_Period => Seconds (1);` — code self documents the cyclic task attributes
- `task body Counter is` — new scope for task statements and declarations
- `Put_Line ("Counter Start");` — task body only contains unique task code
- `J := J + 1;` — cyclic and initialisation code easy to identify
- `end Counter;` — compact

Using Ada 2012 primitives

Listing 35
The periodic task using Ada 2012 primitives.

```
task Counter
   with Priority => 15;

task body Counter is
   Period  : constant Time_Span := Seconds (1);
   Wake_Up : Time := Clock;

   type Private_Natural is new Natural;
   J : Private_Natural := 0;

begin
   Put_Line ("Counter Start");
   Put (J);
   loop
      delay until Wake_Up;
      Wake_Up := Wake_Up + Period;

      J := J + 1;
      Put (J);
   end loop;
end Counter;
```

- `task Counter` — cannot easily see if and what sort of cyclic task
- `with Priority => 15;` — only task priority can appear on a task's definition
- `task body Counter is` — new scope for task statements and declarations
- `Period : constant Time_Span := Seconds (1);` — cyclic task attributes hidden in body of task
- `J : Private_Natural := 0;` — scope mixes declarations to make task cyclic with task specific code
- `begin` — have to parse the task body careful to determine type of cyclic task
- `loop` / `delay until` — needs changing if running on EDF policy
- `delay until Wake_Up;` — scope contains statements to make task cyclic
- `Wake_Up := Wake_Up + Period;` — code has to be repeated for each periodic task
- `J := J + 1;` — cyclic and initialisation code not easy to identify
- `end Counter;` — not much longer than the Cyclic Task Specification but requires more work to read and write

Summary

The *Cyclic Task Specification* provides an implementation that is easy to understand and write: reducing the time spent creating and maintaining cyclic tasks. The approach preserves Ada's task abstraction by providing a new scope for declarations and statements, yet does so without the users having to incorporate any statements or declarations relating to the task's cyclic operation.

Periodic Counter Comparison

Using the Real-Time Utilities Library

Listing 36
The periodic counter using the *Real-Time Utilities Library*.

```
type Private_Natural is new Natural;        -- no scope provided for non-object declarations

type Counter_Task_State is                  -- need to provide a new type for task state
  new Periodic_Task_State with record       -- must select right library type to derive from
   J : Private_Natural := 0;                -- state free of cyclic related declarations
end record;

procedure Initialize (S : in out Counter_Task_State);
procedure Code (S : in out Counter_Task_State);
procedure Overrun (S : in out Counter_Task_State);

Counter_State       : aliased Counter_Task_State;      -- state object accessible by other tasks
Counter_Release     : aliased Periodic_Release         -- only indication task is periodic
                        (S => Counter_State'Access);   -- a task requires three separate object declarations
My_Counter : Real_Time_Task (S => Counter_State'Access,  -- need to choose the right types to use and link
                                                         -- the task objects correctly
                             R => Counter_Release'Access,
                             Init_Prio => Default_Priority);

procedure Initialize (S : in out Counter_Task_State) is
begin                                       -- cyclic task attributes hidden in body of task
   S.Pri    := 15;                          -- user has to set cyclic attributes themselves
   S.Period := Seconds (1);                 -- attributes cannot be made static

   Put_Line ("Counter Start");              -- initialisation statements easy to locate but
                                            -- mixed with task properties
   Put (S.J);
end Initialize;

procedure Code (S : in out Counter_Task_State) is
begin
   S.J := S.J + 1;                          -- state objects referenced via state object
   Put (S.J);                               -- cyclic statements easy to locate
end Code;
-- convoluted and unclear for a simple periodic task
```

Summary (continued)

By contrast, using Ada 2012 primitives requires the user to repeat a basic cyclic task template, risking the introduction of additional faults into the system due to an incorrectly filled template. Likewise, a challenge exists to determine what sort of cyclic task the counter implements at a glance.

The *Real-Time Utilities Library* is complex, requiring three objects and a state type to create a task. The task lacks its own scope, with declarations and the state object accessible by other tasks: exposing information leading to maintenance and reliability problems. Referencing state objects via a master object requires extra typing and creates reliability issues when state and local objects share the same name. Difficult to move existing code to this approach with each state object requiring a prefix. Need body code to determine cyclic attributes.

Box 1

State Communicator Comparison

Using the Cyclic Task Specification

Listing 37
Sporadic state communicator using the *Cyclic Task Specification*.

```
protected Handler_Object is                 -- protected object to provide the exhausted budget handler
   procedure Exhaustion (T : Task_Id);
end Handler_Object;

protected body Handler_Object is
   procedure Exhaustion (T : Task_Id) is     -- handler sharable amongst a number of tasks
   begin
      Log_Budget_Exhaustion (T);
   end Exhaustion;
end Handler_Object;

task State_Communicator                     -- most of the task consists of defining its attributes
                                            -- on the task declaration, which is self documenting
  with Priority              => 15,
       Cyclic_Task           => Sporadic,
       Cycle_Phase           => Milliseconds (200),
       Cycle_MIT             => Seconds (1),
       Early_Event_Response  => Delay;
       Relative_Deadline     => Milliseconds (500),
       Deadline_Response     => Abort_Cycle,  -- choice to abort task or just the cycle
       Execution_Budget      => Milliseconds (100),
       Execution_Response    => Handler,      -- or invoke a handler
       Execution_Handler     => Handler_Object.Exhaustion'Access;

task body State_Communicator is
   Active_Comm   : Boolean := False;
   State_To_Send : PO_State;
begin
   Comm_Device.Set_Up;
cycles
   if Active_Comm then
      Comm_Device.Reset_Channel;
   end if;
                                            -- protected object containing state to communicate
   State_To_Send := The_Protected_Object.Get_State;
   Comm_Device.Send (State_To_Send);
end State_Communicator;

procedure Update_V (V_Value : V_Type) is    -- procedure updating a state variable and
                                            -- then communicating the updated state
begin
   The_Protected_Object.Set_V (V_Value);
   new cycle State_Communicator;            -- new cycle statement takes the name of the task object
end Update_V;
```

State Communicator Comparison

Using the Real-Time Utilities Library

Listing 38
Sporadic state communicator using the *Real-Time Utilities Library*.

```
type State_Comm_State is new Sporadic_Task_State with record
   Active_Comm   : Boolean;
   State_To_Send : PO_State;
end record;

procedure Initialize (S : in out State_Comm_State);
procedure Code (S : in out State_Comm_State);
procedure Overrun (S : in out State_Comm_State);

State_Comm_State_Object : aliased State_Comm_State;
State_Comm_Release: aliased
                 Sporadic_Release_With_Deadline_Miss_And_Overrun
                   (S => State_Comm_State_Object'Access,
                    Termination => True);
State_Comm_Task : Real_Time_Task_With_Deadline_Termination
                     (S => State_Comm_State_Object'Access,
                      R => State_Comm_Release'Access,
                      Init_Prio => Default_Priority);

procedure Initialize (S : in out State_Comm_State) is
begin
   S.Pri                 := 15;
   S.MIT                 := Second (1);
   S.Relative_Deadline := Milliseconds (500);
   S.Execution_Time     := Milliseconds (100);
   S.Active_Comm := False;
   Comm_Device.Set_Up;
end Initialize;

procedure Code (S : in out State_Comm_State) is
begin
   if Active_Comm then
      Comm_Device.Reset_Channel;
   end if;
   S.State_To_Send := The_Protected_Object.Get_State;
   Comm_Device.Send (S.State_To_Send);
end Code;

procedure Overrun (S : in out State_Comm_State) is
begin
   Comm_Log_Budget_Exhaustion;
end Overrun;

procedure Update_V (V_Value : V_Type) is
begin
   The_Protected_Object.Set_V (V_Value);
   State_Comm_Release.Release;
end Update_V;
```


Box 2

State Communicator Comparison

Using Ada 2012 primitives

Listing 39
Sporadic state communicator using Ada 2012 primitives.

```
protected Communicator_Object is
   procedure CPU_Budget_Exceeded (TM : in out Timer);
   procedure Notify_Missed_Deadline (Event: in out Timing_Event);
   entry Missed_Deadline;
private
   Deadline_Missed : Boolean := False;
end Communicator_Object;

protected body Communicator_Object is
   procedure CPU_Budget_Exceeded (TM : in out Timer) is
   begin
      Log_Budget_Exhaustion;
   end CPU_Budget_Exceeded;

   procedure Notify_Missed_Deadline (Event: in out Timing_Event);
   begin
      Deadline_Missed := True;
   end Missed_Deadline;

   entry Missed_Deadline when Deadline_Missed is
   begin
      Deadline_Missed := False;
   end Missed_Deadline;
end Communicator_Object;

task State_Communicator with Priority => 15;

Send_State : Ada.Synchronous_Task_Control.Suspension_Object;

task body State_Communicator is
   MIT               : constant Time_Span := Seconds (1);
   Phase             : constant Time_Span := Milliseconds (200);
   Relative_Deadline : constant Deadline  := Milliseconds (500);
   CPU_Budget        : constant Time_Span := Milliseconds (100);

   Wake_Up          : Time := Clock + Phase;
   CPU_Timer        : Timer  (T => Current_Task'Access);
   Deadline_Timer   : Timing_Event;
   P_Control        : Communicator_Object;
   Timer_Cancelled  : Boolean;

   Active_Comm   : Boolean := False;
   State_To_Send : PO_State;

begin
   Comm_Device.Set_Up;
   Deadline_Time.T := Current_Task;
   Ada.Synchronous_Task_Control.Set_False (Send_State);
```

task specific protected object to provide execution budget and missed deadline handling

a support object required in addition to task object

more cyclic-related declarations to contend with

State Communicator Comparison

```
   loop
      Set_Handler (Event    => Deadline_Timer,
                   At_Time => Wake_Up + Realative_Deadline,
                   Handler => P_Control.Missed_Deadline'Access);
      Suspend_Until_True (Suspend_Until_True);
      delay until Wake_Up;
      Wake_Up := Wake_Up + Period;
      Set_Handler (TM      => CPU_Timer,
                   In_Time => CPU_Budget,
                   Handler =>
                           P_Control.CPU_Budget_Exceeded'Access);
      select
         Notify_Missed_Deadline;
      then abort
         if Active_Comm then
            Comm_Device.Reset_Channel;
         end if;
         State_To_Send := The_Protected_Object.Get_State;
         Comm_Device.Send (State_To_Send);
      end select;
      Cancel_Handler (Event     => Deadline_Timer,
                      Cancelled => Timer_Cancelled);
      Cancel_Handler (TM        => CPU_Timer,
                      Cancelled => Timer_Cancelled);
   end loop;
end State_Communicator;

procedure Update_V (V_Value : V_Type) is
begin
   The_Protected_Object.Set_V (V_Value);
   Ada.Synchronous_Task_Control.Set_True (Send_State);
end Update_V;
```

changing how the sporadic task reacts to early release requires this code to change

implementation requires significant amount code, which is not trivial to change if the task's attributes change

the task's cyclic statements are buried here inside the select statement

not clear from code what sort of cyclic task is implemented

cycle release event signaled via another object, careful naming required to identify which task receives event

Summary

The effort required to implement and understand a sporadic task with deadline and budget enforcement using Ada 2012 primitives demonstrates the needs for an alternative, particularly as a real-time system contain a non-trivial number of cyclic tasks. The *Real-Time Utilities Library* provides one approach, proving more suited to the complex example than the simple periodic counter.

However, the advantages the *Cyclic Task Specification* has in the simple periodic counter example apply here as well: providing reliability, maintainability and productivity benefits over other approaches. Additionally, the approach makes it simple to modify the behaviour of the task by simply changing its aspects, allows the sharing of response handers among different tasks and raising a cyclic event occurs directly on the task object as opposed to a proxy object.

Box 2 continued

Conclusion

This chapter has presented the *Cyclic Task Specification* as the solution to closing the gap between the design and specification of cyclic tasks and their implementation in Ada: by incorporating a cyclic task abstraction within Ada through extensions to Ada's tasking syntax and semantics. Its approach, unlike existing library-based abstraction approaches, provides a natural expression of cyclic tasks within Ada's existing task abstraction. The user no longer needs to consider a cyclic task as a set of components loosely linked together, making it easier to implement and understand a cyclic task expressed in Ada.

The cost of the *Specification's* approach is a larger and more complex language; the introduction of a new keyword further affects existing code. Mitigating the complexity, only a core subset of the cyclic task model needs incorporating within Ada's core language, with the rest of the *Specification* implemented within the *Real-Time Systems Annex*.

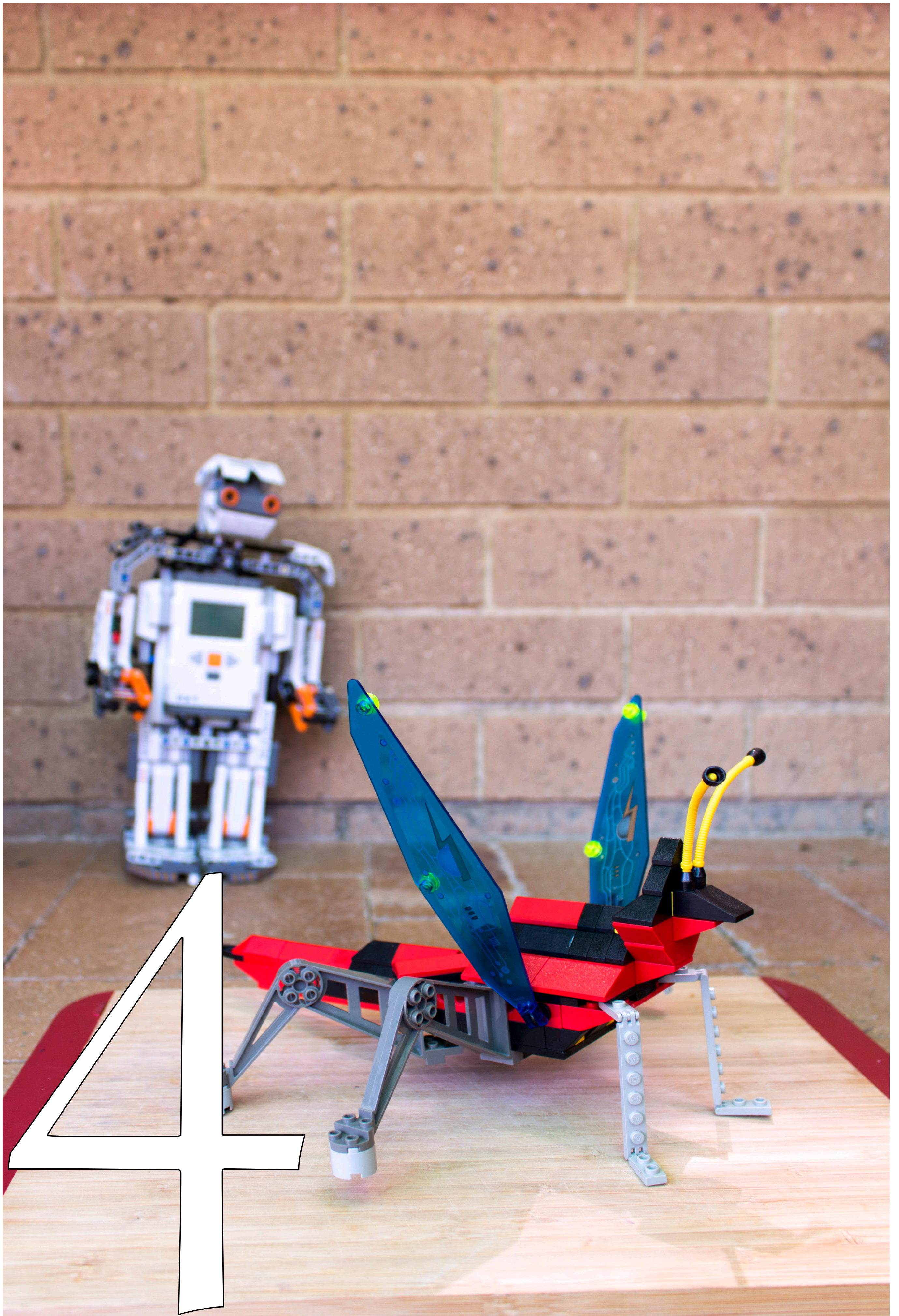
What is lost with a more complex language is gained through the clarity and structure brought to cyclic tasks that is not currently available to users. In turn, software reliability and maintainability improves since there is lower risk of errors being introduced when cyclic tasks are written, interpreted and maintained. Combined with the productivity gains from users spending less time reading and writing tasks, the *Specification* reduces the life-cycle cost of software. Furthermore, the benefits extend to users outside the real-time community since other application domains commonly use cyclic task patterns.

When the user elects to use only static cyclic attributes, the benefits provided by the *Specification* become more pronounced: all these attributes are now neatly listed on the task type's declaration and the compiler ensures these attributes cannot change at run-time. Other users no longer need access to the task's body to determine the cyclic attributes of the task, offering better information hiding. Furthermore, the benefits extend to the software tools that real-time users use to ensure their software meets its specification, opening up interesting possibilities. By defining the static properties of a cyclic task in a standardised approach, software tools can now read and write the properties straight from the code. Tools like schedulability and WCET analysers can use the same semantic information as the compiler, removing an avenue for errors to occur when copying attributes between code and analyser.

Finally, the *Specification* addresses concerns regarding its flexibility and ability to suit the needs of users. In the first instance, the *Specification* provides support for a range of different cyclic tasks and cycle enforcement measures commonly desired in real-time systems. Secondly, with the semantics implemented by the run-time and guided by task attributes, scope exists for Ada environments or the language to provide more types of cyclic task or cycle enforcement in addition to the *Cyclic Task Specification*. This can be done without affecting existing users. Underpinning the ability is the flexibility provided by the core cyclic task model: that of a task which executes its cyclic code in response to a cycle event.

The flexibility offered by the *Cyclic Task Specification* is novel as previous syntax-based cyclic task approaches required new keywords to express new cyclic task properties, which can impact existing users of the language. To back this up, users may still use Ada's low level task features to build their cyclic task if it does not fit within the cyclic task model implemented by their Ada environment,

providing more flexibility than library-based approaches like RTSJ which may not expose task and scheduling behaviour to the user.


4

Chapter 4

Review of GNAT for Bare Boards for Real-Time Systems

GNAT FOR BARE BOARDS (GNAT-BB) IS A HARD REAL-TIME EXECUTIVE FROM AdaCore providing a simplified, Ravenscar Profile compliant Ada run-time. This chapter reviews the design of GNAT-BB and identifies issues detrimental to real-time systems. Motivation for the review lies in whether GNAT-BB, as the only modern Ada real-time executive for microcontrollers, provides a suitable base for implementing the *Cyclic Task Specification*.

Central to GNAT for Bare Boards lies GNAT: an Ada development environment developed by AdaCore. GNAT-BB takes a subset of the GNAT run-time suitable for hard real-time systems and incorporates a small kernel, allowing operation of the run-time on a processor without an operating system (AdaCore, 2015B). GNAT-BB's design permits using microcontrollers like Freescale's MPC5554 and STMicroelectronics' STM32F4 — devices possessing as little as 64KB RAM and 265KB Flash memory. Its minimal nature provides a simple and deterministic system.

However, the simple design adds complexity to the timing analysis of programs built using GNAT-BB and suffers from flaws in the calculation of task execution time. This chapter will explore these issues and their impact on hard real-time Ada programs. First, the chapter will explore the general design of GNAT and GNAT-BB. Secondly, the chapter will examine the defects affecting applications making use of GNAT-BB because of its design.

GNAT

Two features define the GNAT Ada compiler: the software is open-source and rapidly implements the latest Ada standards. The compiler consists of five components (Figure 5): an Ada front-end, the GCC code generator, a binder, a linker and a run-time library called the GNAT Run-Time Library (GNARL) (Comar, Gasperoni & Schonberg, 1994; Dewar, 1994). Combining a portable and configurable run-time library with the code generation support provided by GCC, GNAT provides an Ada environment that supports a large set of different operating systems and processors — regardless of the direct support from GNAT's creator AdaCore.

GNAT for Bare Boards (GNAT-BB) uses the configurable nature of GNAT to provide a run-time subset appropriate for Ravenscar Profile systems, inheriting GNAT's strengths and weaknesses. Availing GNAT's portability, GNAT-BB turns the GNAT run-time into a real-time executive by incorporating a small kernel providing only the services required by the run-time. The focus on GNAT centres on how this flexible run-time enables GNAT for Bare Boards.

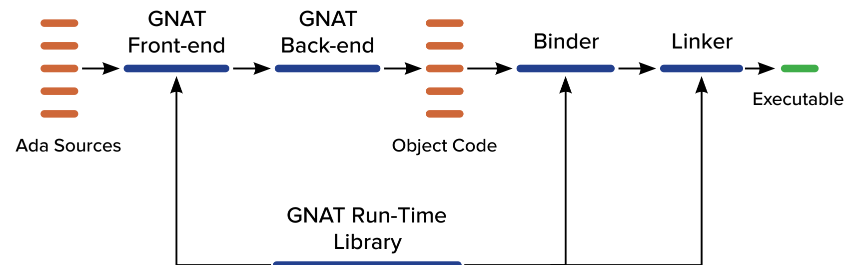


Figure 5
The GNAT compiler chain.

Flexible Run-Time Library

Ada is large language — with optional Annexes — requiring a run-time library to provide language defined library units and support for the language's dynamic semantics. However, Ada programs typically do not use all the core language because parts of the language are not suitable or required as dictated by the design of the application or certification restrictions. In these cases, a desire for a subset of the GNAT run-time may exist: enabling a smaller run-time to run on operating systems that may not support the removed features. As a response, Ada provides a way of restricting its languages features and allows the compiler to specify profiles that collect a set of restrictions (ARM 13.12; Barnes, 1995).

The GNAT Zero Footprint Profile (ZFP) presents an extreme example of a profile: an Ada subset producing a run-time with no memory footprint (AdaCore, 2015c). Excluded from the profile are the dynamic Ada semantics, allowing applications to run without an underlying operating system. Once a program uses dynamic semantics or many of Ada's library units, GNAT requires an underlying operating system to provide the required services: simplify the run-time or because the semantics require interaction with the underlying system, for example dynamic memory allocation or tasks (AdaCore, 2015c). The reliance on underlying operating system services poses a challenge for a portable run-time as operating systems can implement these services in different ways. Even when operating systems provide services in accordance to a known standard, subtle differences can exist between standard implementations.

Consequently, a flexible Ada run-time requires the:

- ability to create subsets of the run-time;
- use of operating system services common across a wide range of multiple operating systems; and
- awareness of the differences in these services.

GNAT achieves this flexibility by: resolving run-time entities at compile time; defining a common low-level interface; and providing target-specific versions of run-time packages.

run-time entities resolved at compile time

As GNAT compiles an Ada package, the compiler front-end inserts run-time entities — constants, subprogram calls and types— into the package's code to realize the desired dynamic semantics (Schonberg & Banner, 1994). GNAT loads the library packages containing theses entities dynamically at compile time using the Rtsfind package (Free Software Foundation, 1992). Part of Rtsfind is a check to ensure the inserted entity and the package housing the entity exists within the run-time library. If the check fails, with GNARL built as a configurable run-time, GNAT raises a compile time error at the point in the user's code where the entity is required.[5]

Thus, Rtsfind enables the safe use of GNARL subsets independent of the Ada front end by discovering what the run-time library supports as the front-end processes the user's code, allowing a single compiler to support multiple run-time subsets (Schonberg & Banner, 1994). GNARL subsets are then simple to create by removing GNARL packages not relevant to the user's application (AdaCore, 2015c).

GNAT low-level library

The ubiquitous nature of C allows GNARL to access many operating system services in a portable, platform independent manner through the C standard library (AdaCore, 2015c). Services include: memory allocation and management, file IO, and maths functions. Missing from the C standard library until its 2011 revision was support for accessing the operating system's task services (ISO/IEC, 2011). While a standard for accessing an operating system's task services exists in the form of POSIX (a standard for a portable operating system interface), unlike the C standard library not every operating system natively supports or prefers POSIX Threads. For example, operating systems like Windows provide their own threading interface while Solaris prefers its own threading library. Even among operating systems supporting POSIX threads, differences exist due to the standard providing room for differing levels of compliance, interpretation and private extension (IEEE Computer Society, 2008).

GNAT Low-level Library (GNULL, formally GNU Low-Level Library) provides GNARL a common, portable interface to an operating system's tasking services (Giering, Mueller & Baker, 1994). Sitting between GNARL and the underlying operating system (Figure 6), GNULL accords an operating system independent set of low-level tasking primitives from which GNARL builds its tasking run-time upon: primitives influenced by the POSIX real-time API (Baker, Oh & Moon,

5. GNAT will throw a fatal error if the run-time is configured as a standard run-time.

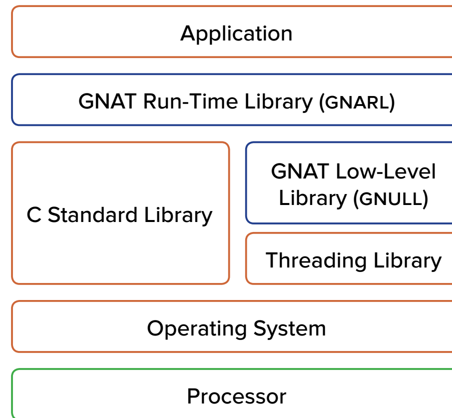


Figure 6 GNAT Run-Time library architecture.

1997). GNULL then maps these primitives to the underlying operating system services. Operating systems implementing POSIX Threads permit a straightforward mapping, with GNULL acting as a lightweight wrapper. Other operating systems, like Windows, require GNULL to perform more work to connect its interface to the underlying services (AdaCore, 2015A). The use of GNULL ultimately constrains the implementation of GNARL's tasking services; whose primitives dictate the strength and weaknesses of resulting tasking services.

target-specific versions of run-time packages

The last component making GNARL a flexible run-time is the capability to provide target-specific packages (Baker & Giering, 1996). For example, the `System.Task_Primitives.Operations` package provides the tasking primitives interface. Each target requires a separate package body mapping these primitives to the underlying target's services as shown in Figure 7.

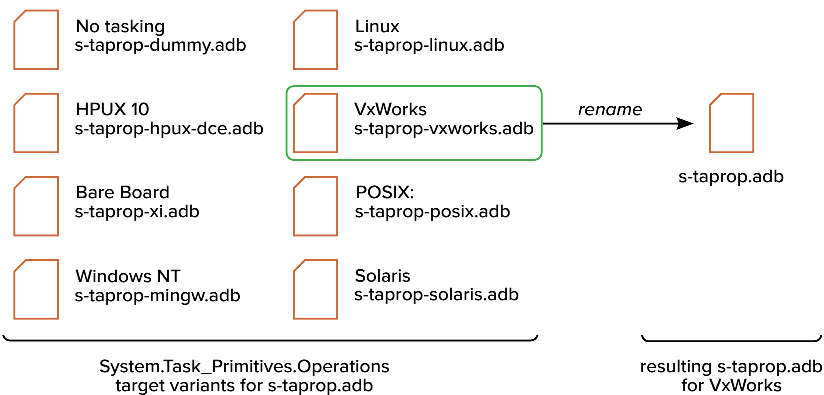


Figure 7 Targeting run-time packages for different systems.

When the run-time is built, the build system renames the corresponding target package to the correct package file name: for example, compiling GNARL for VxWorks would see `s-taprop-vxworks.adb` renamed to `s-taprop.adb`. The file renaming occurs as part of GNAT's build system, which contains the mappings between target packages and their targets (Free Software Foundation, 2014).

Target specific packages delivers GNAT the ability to customise any file making up the run-time — as long the public interface of the package remains unchanged. GNAT typically constrains customisation to library packages connecting the run-time library to the underlying operating system, accounting for the differences in how these systems provide their services.

GNAT for Bare Boards

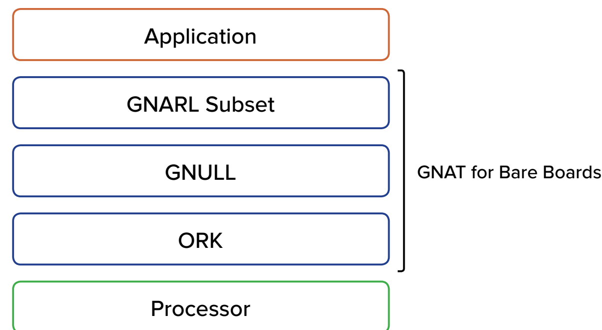


Figure 8
GNAT-BB architecture.

GNAT for Bare Boards (GNAT-BB) provides an Ada run-time that extends the Zero Footprint Profile to support the Ravenscar Profile: a profile offering a deterministic Ada tasking subset (AdaCore, 2015B). GNAT-BB is a unique offering from AdaCore as it combines GNARL with a small kernel providing the required tasking services, allowing it to run directly on a processor without an operating system (Figure 8). This kernel began as the Open Ravenscar Kernel (ORK) by the Real-Time Systems Group at the Technical University of Madrid (la Puente, Ruiz & Zamorano, 2000A).

Altogether, GNAT for Bare Boards supports (AdaCore, 2015B):

- Non-terminating, library-level tasks with static priorities;
- No heap;
- Library-level protected objects with at most one protected entry that has a queue length of one;
- Suspension objects;
- Statically attached interrupt handers;
- Execution time for tasks and interrupt handlers;
- Timing events; and
- Multiprocessors, where task to processor assignment is static.

Kernel Design

The rationale of ORK is simple: providing only the tasking services required by GNAT to produce a Ravenscar Profile run-time runnable directly on a processor (la Puente, et al., 2000B). ORK achieves this by implementing the low-level primitives of GNULL. Supporting only a subset of the Ada run-time simplifies the implementation of these primitives and thus the kernel itself.

GNAT-BB only supports one application, further simplifying the kernel design. Consequently, the kernel uses a single address space with no memory protection, using statically allocated kernel data structures and library-declared objects. Task stacks form part of the kernel data structures representing tasks.

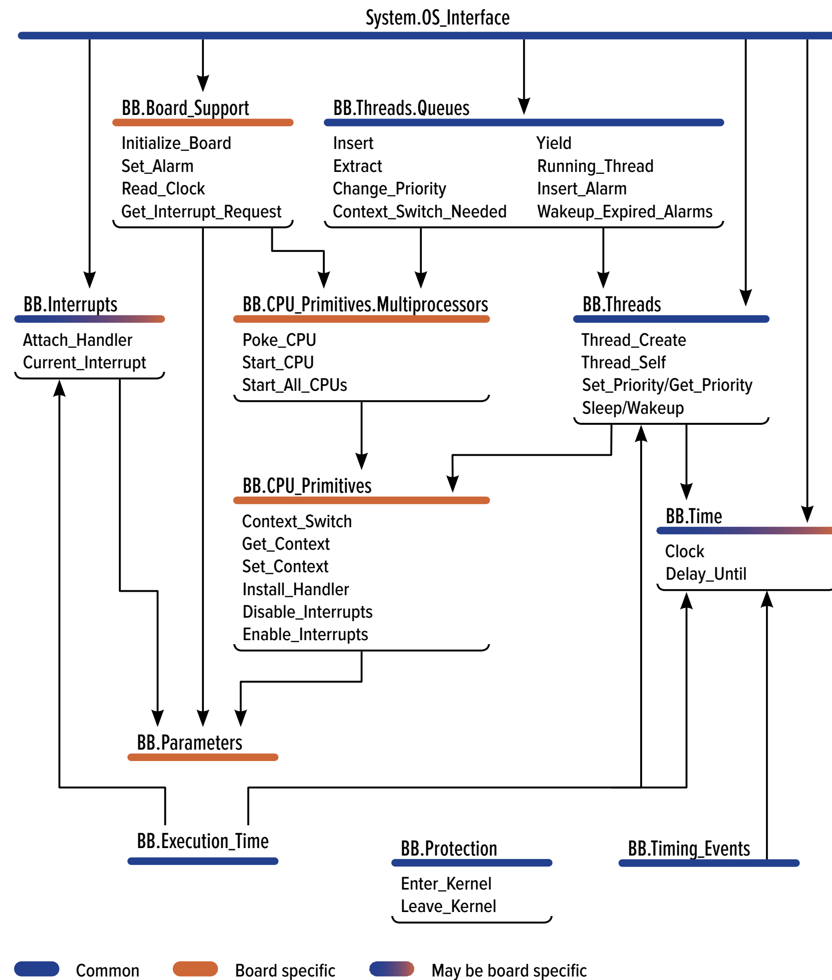


Figure 9
GNAT for Bare Boards core packages with specification dependences and key subprograms.

A single, non-memory protected address space allows a running task to call ORK's kernel operations directly, using the running task's stack space instead of a separate, kernel designated, thread of execution. This approach avoids context switching on every kernel operation, but requires each task to reserve a small part of their stack space for the kernel. To protect ORK's data structures, the kernel is non-preemptive with kernel entry and exit calls wrapping the kernel's public operations.

Eleven core packages form the backbone of the kernel's architecture (Figure 9). The architecture divides platform independent and microcontroller specific operations into separate packages, easing kernel maintenance and portability. Four packages provide the formal interface between the platform independent sections and the underlying processor. Additionally, packages implementing the kernel's time and interrupt support (System.BB.Time and System.BB.Interrupts) may also require processor specific customisation when their internal models differ from the services provided by the processor. GNAT's target-specific package abilities provide the support these processor specific packages.

The platform independent packages provide the services required by GNARL:

- Interrupt management

- Thread management
- Time
- Execution time management
- Timing event management

Thread and time management services deliver the minimal functionality enabling Ravenscar Profile tasks and protected objects, with the remaining services providing kernel support for their respective run-time features.

Task and Protected Object Support

ORK threads represent separate threads of execution within the kernel. Threads are not Ada tasks but instead provide a minimal threading representation upon which GNAT builds Ada tasks (la Puente et al., 2000B). Consequently, an ORK thread only contains information pertaining the thread's state, context, stack, wake time and internal kernel references. This approach permits GNAT-BB to utilise GNAT's existing tasking model: coupling a run-time data structure, an Ada Task Control Block (ATCB), with each system thread (Miranda & Schonberg, 2004). As GNAT-BB provides a restricted run-time, it employs a simplified ATCB with unsupported components and operations removed.

For protected objects, their existing implementation in GNARL relies on locks provided by the underlying operating system (Giering & Baker, 1992). ORK, however, does not provide the required locking facilities and instead leverages the Ceiling Locking Protocol to implement mutual exclusion by raising the priority of a task accessing a protected object to the object's ceiling priority (la Puente et al., 2000B). To resolve the differences between the two approaches, GNAT-BB replaces the GNARL's protected object run-time operations with its own.

Scheduling

ORK implements a fixed priority task dispatcher in the System.BB.Threads and System.BB.Threads.Queues packages (la Puente, Zamorano, & López, 2009). The Threads package provides the public interface to the scheduler, while the Threads.Queues package implements the scheduler queues.

The scheduler consists of three queues (Figure 10): a global list containing all threads, a ready queue and an alarm queue. The global thread list tracks all the threads present in the system and used for debugging purposes, while the other two queues support task dispatching. The ready queue consists of a linked list of tasks ready to run, ordered first by priority and then by insertion time. When dispatched, a task remains on the ready queue while the kernel notes the task in its Running_Thread_Table. The alarm queue comprises of a time ordered linked-list of sleeping tasks waiting for their delay until statements to expire.

ORK supports multiprocessor systems by assigning a ready and alarm queue to each processor (Chouteau & Ruiz, 2011). Thread to processors assignment is static and threads cannot migrate between processor, nor do processors access another processor's queues. The collection of ready and alarm queues form the First_Thread_Table and Alarms_Table respectively.

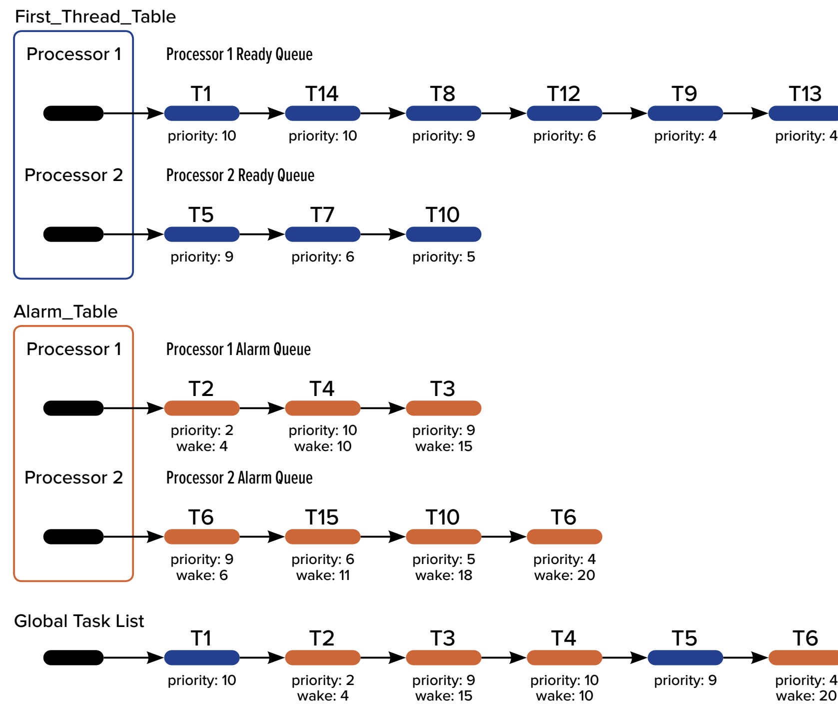


Figure 10
ORK scheduler queues.

Migration of threads between the ready and alarm queues occur when threads sleep and subsequently wake in response to a delay until statement. Threads can also move within a ready queue when a thread changes priority in response to the task exiting a protected object. These sleeping and priority change operations form the public interface to the System.BB.Threads package.

The System.BB.Threads.Queues package contains the internal operations supporting the public threads interface (Figure 11). The package optimises for the Ravenscar Profile by, for example, not supporting the arbitrary removal of threads from queues since the kernel does not require this support. Instead, the package provides operations removing the first thread on each queue (Extract and Extract_First_Alarm). In addition to inserting threads into their correct queue position (Insert and Insert_Alarm), the Queues package also provides explicit Change_Priority and Yield procedures to supply an efficient means of relocating a thread within the ready queue without extracting and inserting the task. Finally, the Wake_Expired_Alarms procedure moves all woken threads from the alarm queue to the ready queue within one procedure.

While System.BB.Threads and System.BB.Threads.Queues manage threads and scheduling queues, task dispatching occurs as part of the System.BB.Protection.Leave_Kernel procedure and at the end of the kernel's processor-dependent interrupt handler wrapper. At these points, the kernel ensures the head of the ready queue is the running thread, performing a context switch if required.

These two points form task dispatching points because they coincide with the exit of the kernel's "protected" state. This state ensures the kernel cannot be pre-empted while modifying its data structures, including the thread queues, to protect the integrity of its data. Because processors only access their respective data structures, disabling interrupts during access gives sufficient protection by preventing a change of context outside the kernel's control. Thus the kernel protects its data structures by requiring all kernel operations that modify them

ORK has four scheduling operations.

Yield, change priority, head ready task to alarm queue, head alarm task to ready queue

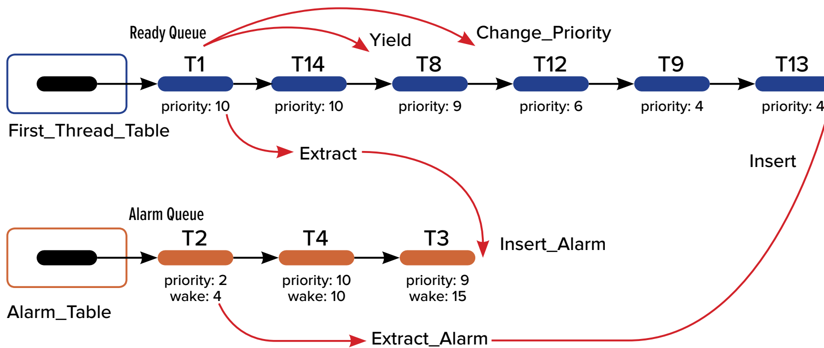


Figure 11
ORK queue operations.

ORK has a task dispatching point at the end of an interrupt handler since the ready queue can change during the handler.

Due to tasks waking or the servicing of queued entry calls.

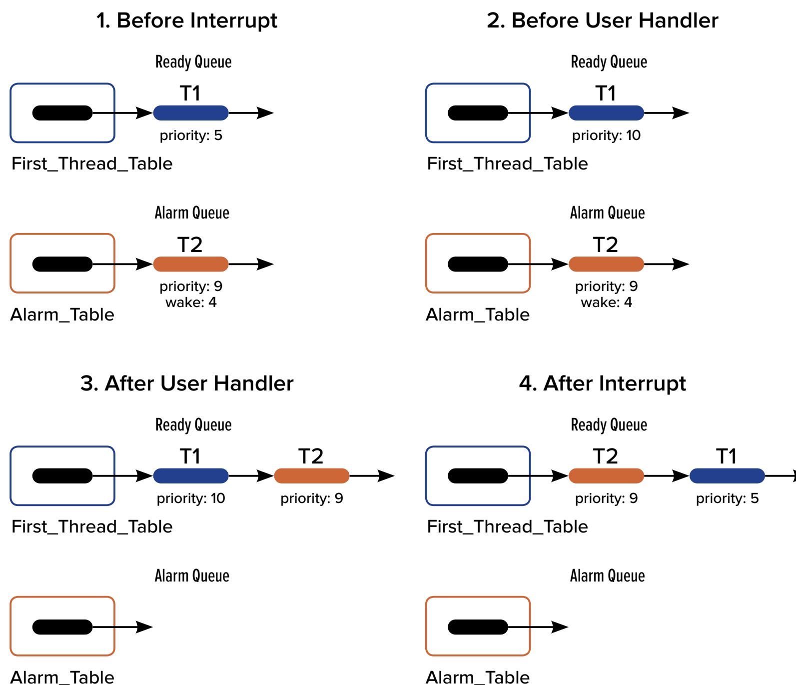


Figure 12
Queues before, during and after an interrupt handler when a task T2 wakes while a lower priority task T1 is servicing the handler. A task dispatching points occurs at the end of the handler due to the priority change.

to use the System.BB.Protection.Enter_Kernel and Leave_Kernel procedures: disabling interrupts between the two calls.

When inside an interrupt wrapper, the processor automatically disables interrupts and thus the kernel exists in a "protected" state without using the System.BB.Protection procedures. However, since the wrapper changes the priority of the interrupted thread — assigning it the interrupt priority — it creates a task dispatching point at the end of the wrapper when the thread's priority restores (Figure 12). At this point, the thread may no longer head the ready queue since an interrupt handler may move another thread to the ready queue with a priority between the interrupted thread and the interrupt priority. Thus, when the wrapper restores the interrupted thread back to its normal priority the kernel switches the context to the new highest priority thread.

Real-Time Problems in GNAT for Bare Boards

GNAT for Bare Boards offers a Ravenscar Profile real-time executive leveraging an existing Ada run-time. At its core is GNARL, a run-time combining a flexible build environment and a tasking model built using common low-level tasking primitives: enabling custom built run-times target different platforms. GNAT-BB provides a purpose-built kernel as a target for GNARL, allowing a simplified GNAT run-time to run without an operating system.

The approach GNAT-BB takes removes the need to maintain a separate run-time. Instead, GNAT-BB benefits from an existing, proven run-time and any benefits derived from its development and maintenance. Additionally, the simple design produces an efficient executive with a small memory footprint. For example, a simple two-task program targeting the STM32F4 contains only 6.1KB of program data and occupies 2KB of RAM.[6]

Unfortunately, GNAT-BB also inherits the same limitations and design compromises as the full GNAT run-time. Significant, and sometimes unintuitive, impediments exist, affecting the schedulability analysis of real-time programs and impeding accurate execution time measurements. The simple design of ORK also complicates the timing verification of tasks, invokes priority inversion and affects the kernel's ability to support low power processor states. The issues inflicting GNAT-BB centre around scheduling, protected objects, execution time and low-power states support.

Scheduling

The alarm queue arranges threads by ascending wake time. Once the wake time of the head task has passed, ORK moves woken threads from the alarm queue to the appropriate position in the ready queue. Depending on timer availability and the implementation of the real-time clock, servicing may occur in response to a timer interrupt — used on SPARC and Power Architecture targets — or as part of a software real-time clock update — used by some ARM architecture targets.

While simple, the servicing occurs regardless of the priorities of the running and woken threads. Using Figure 10 on page 84 for example, the kernel will interrupt the two running tasks each time a lower priority task wakes as shown in Figure 13. This despite all the tasks on the alarm queue not being eligible to dispatch until the running tasks yields the processor. The result causes run-time

A task runs longer than expected due to the kernel servicing waking lower priority tasks.
The interruption is reflected in the task's execution time.

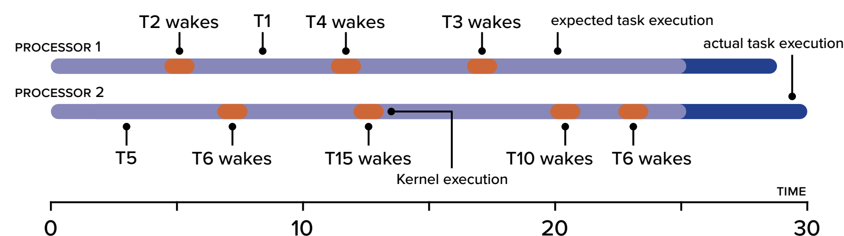


Figure 13
Kernel interruption of the running tasks in Figure 10 on page 84 due to lower priority tasks waking and being moved to the ready queue.

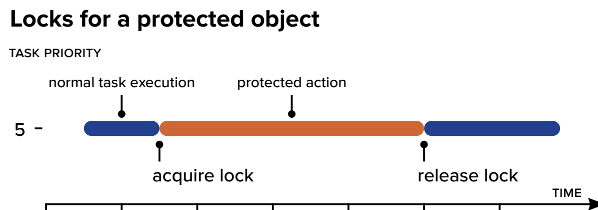
6. Excluding memory reserved for task and interrupt stacks.

priority inversion where the kernel interrupts a high priority thread simply to move a lower priority thread to the ready queue. Correct operation of the system requires the interference to be included in the system's schedulability analysis; otherwise, the higher priority thread could miss its deadline due to the servicing of these lower priority threads — particularly on systems which may have threads waking in clusters. Consequently, while reducing ORK's overheads, the simple treatment of waking threads complicates the schedulability analysis because it has to consider the lower priority threads interference.

Furthermore, the ready and alarm queues insert operations are $O(n)$ bounded. Hence, the worst-case insertion time can grow quickly as the number of threads increase, even if the average insertion times may not.

Protected Objects

Protected objects without entries use a simple model: a task gains access to an object by possessing its associated lock. By using the Priority Ceiling Protocol (PCP), GNAT-BB efficiently emulates locks by raising the accessing task's priority to the priority ceiling of the protected object, as shown in Figure 14. A protected action begins when a task gains the lock and completes once the lock is released (ARM 9.5.1).



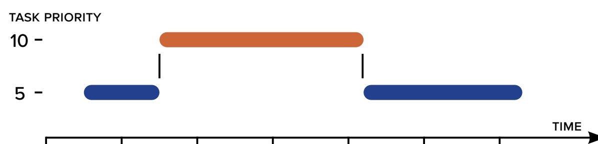


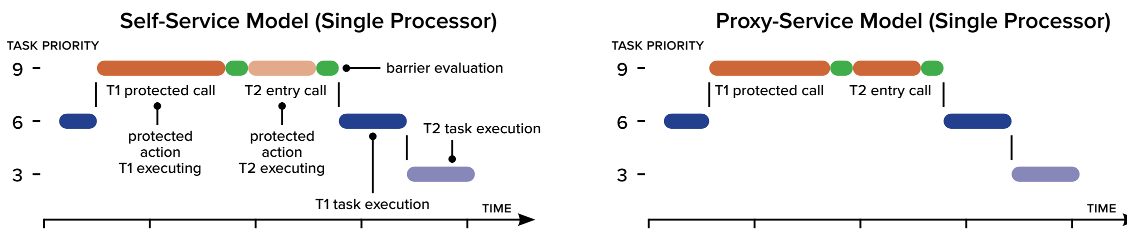
Figure 14
Protected actions for a protected subprogram call. Protection can be implemented using locks or raising the calling tasks priority using PCP.

With entries, the complexity of the protected object model increases. An entry provide the same read-write exclusive access to a protected object as a protected procedure, but with an associated barrier. The barrier opens and closes based on the evaluation of the barrier guard. Normally the guard associates with the protected object's state, necessitating a task to gain the lock of the protected object before evaluating the barrier. An open barrier allows the task to complete the entry call while a closed barrier places the task's entry call on an internal entry queue and sees the task release the protected object's lock.

Entry barriers re-evaluate when protected procedures and entries calls complete but before the protected action ends. If barriers with queued calls open at the end of the protected action, the protected action services the queues until all open entry queues are empty.

Ada's protected object rules do not define who services queued entry calls. An expected servicing model would have tasks service their own calls, under the reasonable assumption a task will always execute its own code, as shown on the left of Figure 15. The alternative is a proxy model where another task

Proxy-service model can be more efficient than the self-service model with the removal of context switches within the protected action.



However, with multiple processors, the proxy-service model holds the first task until the completion of the protected action.

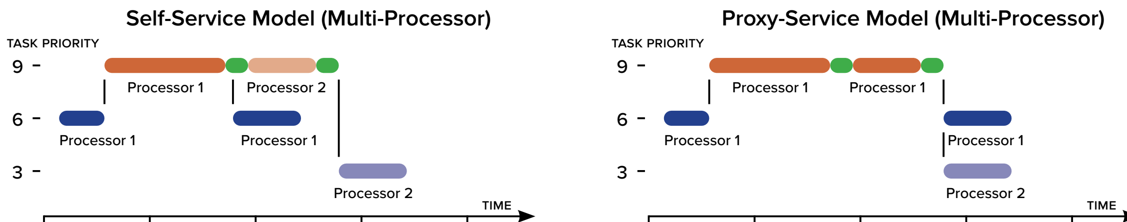


Figure 15

Serving an entry call at the end of a protected action via the self-service and proxy-service methods. Barrier evaluated in existing task context. No other tasks eligible to run. Multiprocessor results holds when T1 and T2 processor assignments differ.

services the entry call on behalf of the caller. The self-service model provides the benefits of:

- being more intuitive;
- the ability to treat the entry call as a procedure call and not having to transfer entry parameters between tasks;
- better utilization on multiprocessor systems, as the leaving task can immediately continue running on another processor;
- ensuring a task only executes its own code; and
- reported task execution time correlating with the code executed.

Despite these advantages, GNAT (and consequently GNAT-BB) use the proxy-service model (Giering, Mueller & Baker, 1993). Here, the task which initiated the protected action services entry calls on open entry barriers until all open entry queues are empty. GNAT uses the proxy-service model as self-service requires the task leaving the protected object to pass the lock it holds to the task whose entry call needs servicing. However, the locks offered by the platforms GNAT supports do not provide such a lock passing mechanism: the lock can only be released, allowing any waiting task to acquire the lock. While workarounds exist, the authors of GNARL found them inefficient compared to the proxy-service model (Giering et al., 1993). Furthermore, the proxy-model suits single processor systems by having entry calls serviced in the same context, removing the context switches required by the self-service model (Figure 15 right).

The proxy-service model also simplifies GNAT's implementation of asynchronous transfer of control (ATC) and timed entry calls, features which bifurcate the execution path of a task (Giering & Baker, 1994). Under the proxy-model, another task will automatically service the queued entry call, allowing the calling task to execute the abortable part of the ATC without creating a new thread.

The time T1 spends inside a protected action depends on other tasks.
Time depends on whether there is an entry call to service.

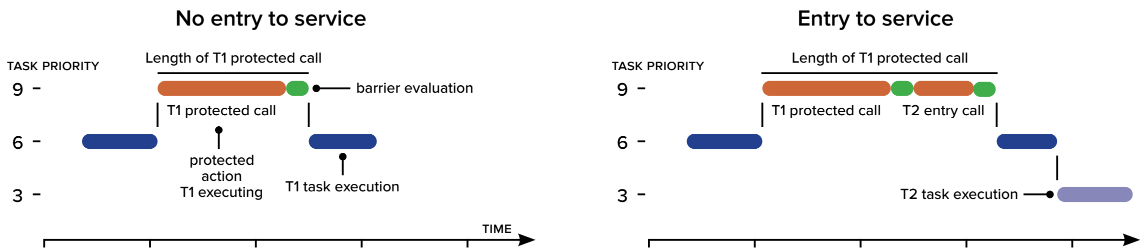


Figure 16
Under the proxy-service model, T1's protected operation length depends on the entry calls that need servicing. The execution time reported by ORK for the right example is 9 for T1 and 2 for T2.

While the proxy-service model suits GNAT's needs, the model negatively affects real-time system development. Protected operations are no longer bounded by the length of the protected operation itself: bounds now have to account for the time spent servicing queued entry calls (Figure 16). Thus, a task's timing analysis requires the entry call timings for each protected object the task accesses and the bound on the number of queue entry calls — unbounded in the general case. Ultimately, schedulability analysis becomes complex.

As a result, the worst-case execution time (WCET) for tasks calling protected objects with entries becomes unnecessarily pessimistic: every protected object access now includes the time spent servicing the worst-case set of queued entry calls. Pessimistic WCET reduces processor utilization and possibly increases costs if the system requires more processing resources to meet system timing requirements. The proxy-model also effects utilization and timing requirements on multiprocessors since it precludes the leaving task from executing outside of the protect object on another processor while the protected action continues (see the multiprocessor examples in Figure 15).

Using the Ravenscar Profile enables GNAT-BB to curb some WCET and schedulability analysis complexity under the proxy-service model. Since the profile restricts each protected object to one entry and one queued entry call, a task will only need to service at most one entry call each protected action.

Execution Time

ORK supports monitoring task and interrupt execution time, exposing the functionality via the Ada.Execution_Time package (Zamorano, Alonso, Pulido & la Puente, 2004). However, the reported execution time includes not only the time a task spends executing, but also the time spent by the kernel while using the task's context. This time includes ORK moving woken tasks to the ready queue (for example Figure 13 on page 86) and while ORK waits for a protected object's processor lock on multiprocessor systems.

Additionally, the proxy-service model for protected objects complicates tracking a task's execution time. Since ORK tracks the execution of a running task, a task's execution time will include the time spent executing its own code and the time spent servicing entries for other tasks (Figure 16). The reverse also occurs: the calling task does not get credited for the entry call service time if another task services the call.

Together, these issues mean GNAT-BB may report an execution time larger than the calculated execution time bound for the task unless included as part of the analysis. Difficulties exist though in incorporating these extra time sources: the time spent in the kernel requires prior knowledge of how many tasks will

wake while the task executes and the bound on the waiting time for a protected object processor lock. While the proxy-service model simplifies under the Ravenscar Profile, it still results in pessimistic WCETs: each protected operation includes the entry call servicing time regardless of whether servicing occurs, compounded as protected procedure calls typically outnumber entry calls.

Finally, while GNAT-BB provides the ability to track the execution time of tasks, the Ada executive does not provide execution time timers because the Ravenscar Profile prohibits them (ARM D.13). Consequently, GNAT-BB cannot support execution budgets nor execution time-based execution servers.

Support for Processor Low Power Modes

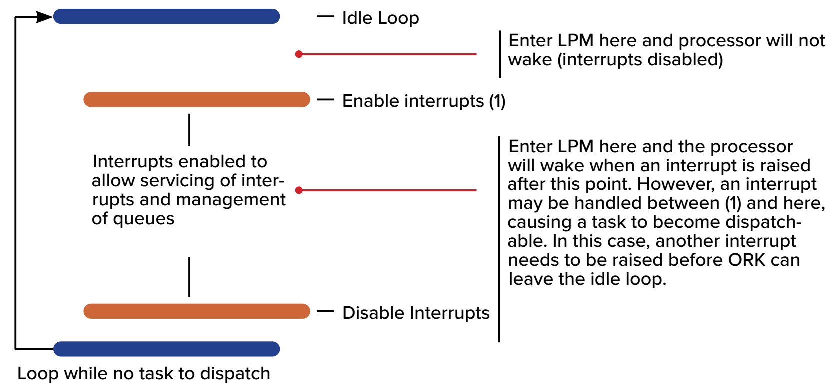


Figure 17
ORK's idle loop, possible points for entering a Low Power Mode (LPM) and the problems with these points.

When the real-time executive has no task to dispatch, GNAT-BB's design prevents its from using the low power modes offered by microcontrollers: instead ORK enters a busy-wait idle loop until a task becomes available, leading to higher power consumption. Consequently, GNAT-BB lacks suitability for systems with low processor utilisation and a requirement to minimise system power — for example, battery operated systems.

ORK's inability to use low power modes occurs because the idle loop occurs within the "protected" mode of the kernel, where interrupts are disabled. However, tasks migrate from the alarm queue to the ready queue (and thus are dispatched) when a time source raises an interrupt. Consequently, to exit the idle state and dispatch a woken task, the kernel enables interrupts for a brief window each loop cycle (Figure 17). The interrupt window also allows the servicing of other interrupts while in the idle state.

The opening of the interrupt window conflicts with the ability to use low power modes. Entry into these states occurs unconditional though a machine instruction or a write to a memory register. When any interrupt occurs, the microcontroller wakes and continues running. Thus, if the microcontroller enters a low power mode before enabling interrupts, the processor core will never wake. On the other hand, entry to a low power mode after enabling interrupts can allow the handling of any pending interrupt in between the enablement and the execution of the low power mode command (Figure 17). Handling the interrupt may result in a thread dispatched. When the context returns to the first thread, after the thread wakes and heads the ready queue, the task will enter the low power mode since the thread's context is still within the busy-wait loop and the processor will not progress until the raising of another interrupt.

Conclusion

GNAT for Bare Boards pairs a Ravenscar Profile subset of the GNAT run-time with a small kernel providing the run-time's required services. GNAT-BB's approach rests upon GNAT's portable, modular and flexible design. The result produces a small real-time executive with low run-time overheads elegantly supporting bare-board Ravenscar Profile applications while maintaining a common code base with the full GNAT run-time. The approach allows GNAT-BB to accrue the benefits from GNAT's development, maintenance and stability.

Deriving a bare-board Ravenscar Profile from GNAT compromises the resulting run-time, as the inherited tasking model relies on threading services inspired by POSIX. Compromises results from GNAT's need to support a wide range of operating systems using POSIX or POSIX-like threading services —most, if not all, operating systems. Consequently, GNAT services queued entry call using the proxy-service model, one not suited to real-time systems as it complicates the timing analysis of tasks and system schedulability analysis. Furthermore, the proxy-service model forces a more pessimistic WCET bound, reducing processor utilisation. Finally, the execution time reported for a task on GNAT-BB includes the time the task spends servicing the entries of other tasks: creating problems where tasks run longer than expected.

GNAT for Bare Boards' simple kernel also poses a number of potential problems for real-time users. For some, the lack of support for processor low power modes when idling creates an issue for systems required to conserve energy.

A larger problem rests with run-time priority inversion: where the kernel interrupts a running task to move woken lower priority tasks to the ready queue. Moreover, the running task's execution time includes the time spent moving these tasks. The effective result causes a lower priority task to consume the running task's resources and potentially cause the task to miss its deadline. A workaround exists if the system's schedulability analysis and the task's WCET bound factors in the interruption, but the solution complicates analysis: particularly as WCET determination now includes activities from outside the task.

Finally, GNAT for Bare Boards simplicity and compactness derives from supporting only the Ravenscar Profile subset of the Ada tasking runtime. While valuable for applications that target the profile, many tasking features outside the Ravenscar Profile are still valuable to real-time users that GNAT-BB is unable to support, for example execution time timers or different task dispatching policies. While some features, like execution time timers, are implementable on top of the current kernel, others require structural changes to ORK to implement due to the Ravenscar Profile guiding the kernel's design. For example, implementing different task dispatching policies is difficult as the kernel is designed around a particular fixed-priority scheduler implemented in the `System.BB.Threads.Queues` package.

Thus, while GNAT for Bare Boards does provide a small and simple real-time executive, its simple nature and a run-time designed to operate on top of a general operating system complicates the timing and schedulability analysis of real-time systems. GNAT-BB's design also limits its ability to support other Ada tasking features that are outside the Ravenscar Profile, but are useful or desired by some real-time users. The problems encountered raise the question whether a real-time executive using a purpose-built tasking run-time better suits the needs of bare-board real-time users.

LADY DENMAN DRIVE
Acton
City
Black Mountain
Botanic Gardens

TOURIST DRIVE
2

TUGGERANONG PWY
Belconnen
Woden
ational Arboretum
Canberra

Chapter 5

Acton

A Real-Time Executive for Ada

Acton is a small real-time executive for ada programs running on microcontrollers. Consisting of a purpose-built Ada tasking run-time and a kernel tailored towards Ada's tasking semantics, the design of the executive balances portability and size while being flexible to support a range of microcontrollers and language features. Acton supports the Ravenscar Profile tasking subset while providing a platform for greater Ada tasking support and experimentation of new features like cyclic tasks. Compilation of Ada programs targeting Acton relies on a modified GNAT compiler called GNAT for Acton.

Beyond implementing the Ada tasking subset defined by the Ravenscar Profile, Acton supports:

- Protected objects with multiple entries, arbitrary length entry queues and non-simple entry barriers
- Tagged records
- Cyclic tasks
- Multiple task dispatching policies

Notable language features not currently supported by Acton include:

- Exception propagation
- Heap allocation
- Asynchronous task communication
- Task rendezvous
- Task termination
- Object initialisation and finalisation

This chapter describes the design of Acton as an Ada real-time executive. Beginning with the rationale behind Acton, the chapter explores Acton's unique architecture focusing on its separation of concerns, key data structures and the implementation of executive operations. The chapter covers in detail the following parts of Acton:

- Kernel resources
- Kernel run-loop
- Scheduling
- Task management
- Protected objects
- Interrupt handling

Rationale

The motivation driving a new Ada real-time executive lies on two fronts: a means to realise cyclic tasks and to address the identified real-time problems within an existing real-time executive (GNAT for Bare Boards). Focusing development on a real-time executive, rather than an Ada run-time and operating system amalgam, enables the use of cyclic tasks for real-time applications using microcontrollers: an application domain typically unable to utilise an operating system and which would welcome cyclic tasks. Additionally, many of GNAT-BB's problems derive from the full GNAT run-time, for example the use of the proxy-service model for protected objects (Chapter 4). As such, little contrast exists between GNAT's real-time executive and GNAT's operating system based Ada run-time for hard real-time Ada programs.

GNAT makes a new real-time executive for Ada applications feasible with the compiler and run-time source code released under the GNU Public Licence (Comar, Gasperoni & Schonberg, 1994). The source code availability facilitates the reuse of proven run-time and compiler components, allowing the executive's development to focus on the aspects making it unique. Access to the compiler source also opens the way to change the implementation of GNAT's tasking facilities and to implement new Ada features like cyclic tasks.

Thus, GNAT's openness and flexibility provides the latitude to create a new real-time executive with a matching Ada run-time unencumbered by the constraints imposed by existing operating systems. Instead, the real-time executive has the freedom to realise Ada's tasking semantics in a natural manner.

Acton embraces the freedom offered and incorporates a subset of Ada's tasking semantics suitable for real-time systems into its kernel. Furthermore, the design of Acton facilitates a real-time executive with flexibility to support Ada tasking semantics outside of the Ravenscar Profile and enables the incorporation of new tasking features.

Consequently, Acton and GNAT-BB possess contrasting design philosophies. GNAT-BB leverages a proven Ada run-time and optimises around the tasking constraints the Ravenscar Profile imposes (la Puente, Ruiz & Zamorano, 2000). Acton on the other hand seeks flexibility to support a wider range of tasking semantics and to be free of the constraints existing operating systems impose. In doing so, Acton resolves the real-time issues present within GNAT-BB.

Another benefit of Acton's approach is support for multiple task dispatching policies — including FIFO_Within_Priorities and EDF_Across_Priorities. Only one other Ada run-time environment supports EDF_Across_Priorities: MaRTE from the University of Cantabria (Rivas, Harbour & Ruiz, 2009) — whose footprint prohibits its use on microcontrollers. Supporting multiple task dispatching policies aids not only real-time users but also demonstrates how cyclic tasks can run transparently under different policies.

Architecture

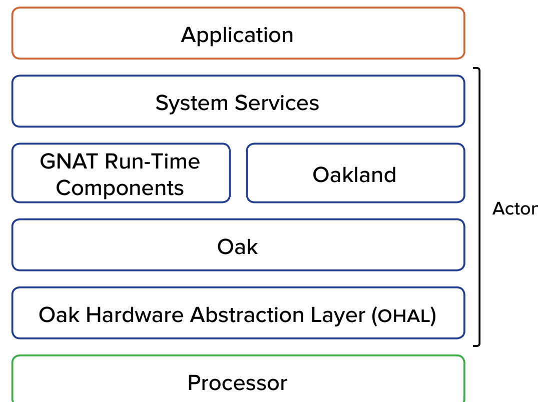


Figure 18
Acton's architecture.

Acton sets out to develop a small, efficient, flexible, portable and predictable real-time executive. However, these properties can conflict and require compromises in one property to achieve the desired outcome in another.[7] Behind predictability, Acton values portability and flexibility. Its architecture (Figure 18) mirrors these priorities and composes of four distinct layers: a kernel, a hardware abstraction layer, a run-time library and system services.

7. For example, the compromises made by Liedtke (1995) to achieve a high performance microkernel where portability is sacrificed for performance

Oak

Oak is Acton's non-preemptive microkernel. Unlike the kernel in GNAT-BB, Oak has its own thread and stack, requiring a context switch to the kernel for all kernel operations. A separate thread allows Oak to support virtual memory and to enforce memory protection between tasks, with Acton designed so only the kernel modifies its internal data structures. While designed to support virtual memory, Oak currently implements a single global address space to simplify the kernel's implementation. Using a separate thread also removes the need for each task to reserve a section of its stack space for the kernel.

Supporting Acton, Oak provides the executive with key tasking services:

- Task support
- Task dispatching
- Interrupt handling
- Timer management
- Inter-task communication

The design of these services draw from Ada 2012's tasking semantics and allow Oak to natively support them, differentiating Oak from other microkernels. For example, protected objects form the backbone of Oak's inter-task communication facilities, with support built into the kernel. It is unique approach because existing microkernels use message passing, shared memory or both as their inter-task communication medium (Elphinstone & Heiser, 2013; Fraser et al., 2004; Liedtke, 1996). In addition to protected objects, Oak has the potential to expand its inter-task communication to include extended task rendezvous as Acton grows to support more of Ada 2012 tasking features.

While the focus on delivering Ada 2012's semantics in the kernel makes Oak larger than most microkernels, the run-time library required to support Ada programs shrinks, enabling more efficient implementation of Ada's tasking features.

Oak does not provide two notable services: heap allocation and scheduling services. The lack of heap allocations follows the recommendation for high integrity systems in ISO/IEC TR 15942 (2000) and mirrors the support provided by GNAT-BB. Like GNAT-BB, memory allocation occurs statically at the library level or on the task stack. An exception lies with the tasks stacks themselves, which Oak allocates from a task stack pool during task creation. With tasks only created statically under the Ravenscar Profile and unable to terminate, dynamic allocation of task stacks does not conflict with high integrity requirements because memory allocation occurs during system initialisation.

While responsible for dispatching tasks, the kernel does not schedule tasks itself. Instead, Oak moves the responsibility of scheduler queue ownership and management to external scheduling services. Oak's approach provides Acton greater flexibility in supporting different task dispatching policies since Oak does not assume a particular set of policies or queue data structures. Furthermore, Oak's approach to scheduling allows nestable scheduling services, with the ability to provide nested schedulers with an execution budget shareable among the tasks it manages.

Oak Hardware Abstraction Layer

Oak's hardware abstraction layer (OHAL) consists of three packages:

- Core Support Package (CSP)
- Processor Support Package (PSP)
- Project Support Package (PJSP)

A family of microcontrollers often builds around a common processor core; for example, Freescale's E200Z6 Power Architecture core forms the basis of six microcontrollers from Freescale's Qorivva family. The OHAL adopts a similar approach by separating core and processor (or microcontroller) specific functionality. As a result, OHAL permits a single, validated CSP across a microcontroller family. This approach simplifies porting Oak to a new microcontroller using a supported core and limits the scope for the introduction of faults.

Specifically, the CSP provides Oak a common interface to a processor's:

- Time services
- Task support services
- Core interrupt services
- Call stack services

Simpler, the PSP currently provides Oak a common interface to the interrupts originating from outside the core and, for some microcontrollers, time services. An external interrupt controller normally manages external interrupts,with the controller differing among microcontrollers sharing a common core. The PJPS on the other hand provides project-defined limits on Oak's internal resources, and specifies processor's clock speed and default stack sizes for Oak's threads.

The flexibility of OHAL enables Acton to currently run on Freescale's MPC5554, Atmel's AT91SAM7S and STMicroelectronics' STM32F4; providing support across two different processor architectures. An early revision of Acton even supported Atmel's 8-bit AVR microcontrollers, though the lack of a suitable real-time clock source makes the AVR ill-suited for real-time applications.

Acton Run-Time Library

Acton's run-time consists of two libraries: GNAT and Oakland. The GNAT Library consists of the GNAT Compiler and Run-Time Components, providing the compiler and run-time support for GNAT's non-tasking Ada language features. The GNAT Library provides proven components not part of the novel aspect of Acton's design — for example tagged records and compiler support.

Oakland provides Acton's tailored run-time facilities, consisting primarily of an Ada tasking run-time. Since Acton implements Ada's tasking semantics predominately in Oak, the implementation of Oakland's run-time interface typically consists of system calls to the tasking services provide by Oak.

System Services

System services provide key system services outside the kernel. Acton provides one service: scheduling. Scheduling on Acton, specifically the ownership and management of task queues, occurs through entities called scheduler agents which have their own individual thread of execution outside the kernel. Each scheduler agent owns and operates its own set of ready and sleep queues. To dispatch a task from its ready queues, the scheduler agent informs Oak of the task and leaves the actual dispatching to the kernel.

Acton's scheduling services provides the flexibility to permit different scheduler agents to operate distinct priority ranges and to allow scheduler agents to schedule other scheduler agents. In the latter case, a nested scheduler agent can have an attached execution budget, with the budget consumed by the tasks the agent has dispatched. When the budget exhausts, Oak will no longer dispatch tasks nominated by the scheduler agent. User selection of non-nested scheduler agents occurs through Ada's Task_Dispatching_Policy and Priority_Specific_Dispatching pragmas, while Acton's execution servers use nested scheduler agents.

Oak Resources

The responsibility of Oak lies in providing Acton's key tasking, interrupt, timer and inter-task communication services. Four kernel resources underlie these services: agents, brokers, Oak Messages and Oak Timers. Agents represent all runnable resources within Acton, such as tasks, schedulers and the kernel itself. Brokers provide Oak's inter-task communication support, currently supporting protected objects. Communication between agents and the kernel occurs through Oak Messages, while Oak Timers provide the kernel's timing facilities.

Agents

Agents represent Acton's runnable resources in Oak, with each agent a thread of execution. Acton uses the term agent rather than thread or task to describe these runnable resources to prevent confusion with Ada tasks (simply called tasks in Acton) and the ambiguity around the term *thread*. All agents derive from a common Oak_Agent type providing the basic functionality required for scheduling and dispatching (Figure 19). Four kinds of primary agents exist:

- Kernel agents
- Scheduler agents
- Task agents
- Interrupt agents

Respectively, each agent provides the kernel resources for the kernel, scheduler services, tasks and interrupt handling. Each primary agent has a different role within the kernel, mandating the use of derivative types to distinguish between the agent kinds and to store kind-specific data. Acton also provides a secondary agent called the sleep agent to provide itself with an idle thread. The sleep agent dispatches when Oak has no other agent to dispatch and only one such agent exists in a given system. The sleep agent does not derive from the Oak_Agent type but uses it directly due to its simplicity.

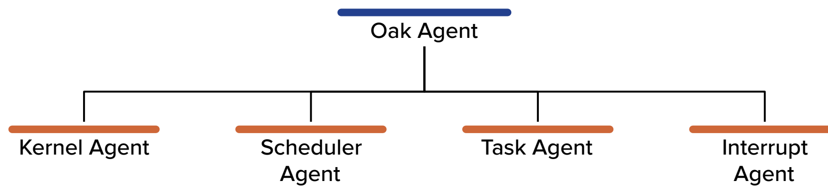


Figure 19
Oak Agent hierarchy.

Each agent has a corresponding kernel data structure, representing the agent within the kernel. The data structure (or kernel object) embodies the state of the agent relevant to the kernel's operation, including for example agent priority and wake time. Only kernel agents may modify kernel objects and then only by the kernel agent responsible for the corresponding agent. Oak's restriction ensures kernel object consistency and enables Oak to perform any operation on its kernel objects without contention. The restriction also permits Acton in the future to protect these data structures from malicious or accidental modification through hardware-based memory protection.

Initially, the primary agents assumed the hierarchy in Figure 19 using tagged records. An Oak_Agent type provided the parent tagged record for the four primary agent kinds, with the sleep agent directly using the Oak_Agent type.

In the end, Oak takes a different approach: using separate record types for the common Oak_Agent type and agent specific components; forgoing the use of tagged records. A primary agent thus consists of two record types: an Oak_Agent record type and one of the following primary agent record types:

- Kernel_Agent
- Scheduler_Agent
- Task_Agent
- Interrupt_Agent

For example, a task agent comprises of an Oak_Agent object and a Task_Agent object — the latter containing just the task agent specific information. Both objects share a single identifier called an Oak_Agent_ID, tying the objects together and enabling access to both objects via a single reference.

Separate record types for the common and kind-specific agent components enable dedicated storage pools for each record type. These storage pools enable Oak to dispense of memory pointers as a means of referencing data within the pools, while still permitting a general identifier to reference any kind of agent.

Oak implements these dedicated storage pools using arrays, indexed via the Oak_Agent_ID type as shown in Figure 20. The Oak_Agent_ID subdivides into a series of subtypes called primary-IDs corresponding to each primary agent kind, with a primary-ID indexing its respective storage pool. This arrangement enables an Oak_Agent_ID to simultaneously reference the Oak_Agent and primary agent objects.

The first value of Oak_Agent_ID, however, does not belong to a subtype but instead forms the kernel's No_Agent identifier: indicating the absence of an agent. The No_Agent also doubles as Acton's sleep agent, conveniently enabling

Figure 20
Agents, and their identification and storage

Oak to dispatch the No_Agent when there is no agent to dispatch. Additionally, collocating with the sleep agent sees the storage reserved for No_Agent productively used. Finally, to allow use of the No_Agent with the primary-ID subtypes, Oak provides *_With_No subtypes combining the primary-IDs ranges and the No_Agent identifier. Primary-IDs and primary-IDs with the No_Agent subtypes have different roles: the latter facilitates agent references while agent specific operations use the former because the No_Agent lacks the agent specific component. Listing 40 presents Oak's definition of the Oak_Agent_ID and its subtypes, with number of agents in each agent kind defined within the PJSP.

Aside from indexing the storage pools, the primary-IDs allow the kernel to identify an agent kind straight from the agent ID and provide a means for the kernel to restrict operations to a particular primary agent. Primary-IDs similarly help the development and maintenance of the kernel; clarifying the kind of agent a variable stores or a subprogram operates on.

The dual object approach for agents allows efficient memory use and facilitates the removal of pointers from within the kernel. Dispensing pointers removes issues around null or invalid pointers — a valid agent ID will always point to an agent data structure. Furthermore, without pointers Oak can in the future switch from Ada to Spark as its implementation language, opening the way for formal verification of Acton's kernel. Finally, not using pointers as agent identifiers can reduce the memory footprint of Oak's data structures, as smaller integer data types can represent Agent IDs. For example, if Oak supports less than 256 agents, the Oak_Agent_ID can fit inside an eight-bit byte as opposed to a four-byte pointer on a 32-bit processor.

Replacing pointers with index types and array-based storage pools does not work with a tagged record type approach, which is why Oak uses dual objects. While a tagged record approach entails one object per agent — a potential benefit — the approach demands separate storage pools for each agent kind. It then requires the general Oak_Agent_ID to index across several different storage pools, thus imposing an extra lookup step when resolving an Oak_Agent_ID. An alternative, using a single pool consisting of a variant record type covering all agent kinds, is too unwieldy to write and maintain.

Oak's approach using fixed arrays as storage pools does have its drawbacks, with the number of agents supported by Oak frozen after compiling the kernel. Likewise, agent object allocation occurs at the same point. Consequently, using less than the maximum number of agents wastes memory: a problem on embedded systems with tight memory requirements. Oak's solution requires a custom kernel built for each project, with a project setting an appropriate number of agents through its PJSP. While at first glance a hassle, the approach does not overly burden embedded systems designers since they typically customise their run-time to include the only the components their projects need

Listing 40
Oak.Agent
oak-agent.ads

```
package Oak.Agent with Pure is
   package PJSP renames Oak.Project_Support_Package;

   type Oak_Agent_Id is mod PJSP.Max_Oak_Agents;

   No_Agent    : constant Oak_Agent_Id := Oak_Agent_Id'First;
   Sleep_Agent : constant Oak_Agent_Id := No_Agent;

   Kernel_Id_Low_Bound  : constant := Oak_Agent_Id'Succ (Sleep_Agent);
   Kernel_Id_High_Bound : constant := Kernel_Id_Low_Bound
                                  + PJSP.Max_Kernel_Agents - 1;

   Interrupt_Id_Low_Bound  : constant := Kernel_Id_High_Bound + 1;
   Interrupt_Id_High_Bound : constant := Kernel_Id_High_Bound
                                        + PJSP.Max_Interrupt_Agents;

   Scheduler_Id_Low_Bound  : constant := Interrupt_Id_High_Bound + 1;
   Scheduler_Id_High_Bound : constant := Interrupt_Id_High_Bound
                                        + PJSP.Max_Scheduler_Agents;

   Task_Id_Low_Bound       : constant := Scheduler_Id_High_Bound + 1;
   Task_Id_High_Bound      : constant := Scheduler_Id_High_Bound
                                        + PJSP.Max_Task_Agents;

   --  Primary-ID defintions.

   subtype Kernel_Id                is Oak_Agent_Id
     range Kernel_Id_Low_Bound    .. Kernel_Id_High_Bound;

   subtype Interrupt_Id             is Oak_Agent_Id
     range Interrupt_Id_Low_Bound .. Interrupt_Id_High_Bound;

   subtype Scheduler_Id             is Oak_Agent_Id
     range Scheduler_Id_Low_Bound .. Scheduler_Id_High_Bound;

   subtype Task_Id                  is Oak_Agent_Id
     range Task_Id_Low_Bound      .. Task_Id_High_Bound;

   --  Primary-ID defintions that include the No_Agent

   subtype Kernel_Id_With_No is Oak_Agent_Id
     with Static_Predicate => Kernel_Id_With_No in No_Agent | Kernel_Id;

   subtype Interrupt_Id_With_No is Oak_Agent_Id
     with Static_Predicate =>
                     Interrupt_Id_With_No in No_Agent | Interrupt_Id;

   subtype Scheduler_Id_With_No is Oak_Agent_Id
     with Static_Predicate =>
                     Scheduler_Id_With_No in No_Agent | Scheduler_Id;

   subtype Task_Id_With_No is Oak_Agent_Id
     with Static_Predicate => Task_Id_With_No in No_Agent | Task_Id;
end Oak.Agent;
```

and they know the number of tasks the project requires. The ability of the approach to facilitate formal verification of the kernel after transitioning to Spark outweighs the cost of having to re-compile Oak.

A description of the four primary agent record types follows.

oak agent
Oak.Agent.Oak_Agent

```
type Oak_Agent_Record is record
   --  Name Information
   Name                    : Agent_Name;
   Name_Length             : Agent_Name_Length;

   --  State Information
   Call_Stack              : Call_Stack_Handler;
   State                   : Agent_State;
   Agent_Interrupted       : Boolean;

   --  Scheduling Information
   Current_Priority        : Any_Priority;
   Scheduler_Agent         : Scheduler_Id_With_No;
   Wake_Time               : Oak_Time.Time;
   Absolute_Deadline       : Oak_Time.Time;

   --  Execution Time Support
   Total_Execution_Time    : Oak_Time.Time_Span;
   Max_Execution_Time      : Oak_Time.Time_Span;
   Current_Execution_Time  : Oak_Time.Time_Span;
   Remaining_Budget        : Oak_Time.Time_Span;
   Execution_Cycles        : Natural;
   When_To_Charge          : Charge_Occurrence;

   --  Internal Support
   Agent_Message_Address   : Address;
   Next_Agent              : Oak_Agent_Id;
   Next_Charge_Agent       : Oak_Agent_Id;
end record;
```

Listing 41
Oak.Agent.Oak_Agent
oak-agent-oak_agent.ads

The Oak_Agent type (Listing 41) provides the common agent components required by Oak to enable the dispatching, execution time tracking and scheduling of agents. Its components fall into five groups:

Agent Name An agent can store a text string containing its name, typically created by the compiler, to allow identification at run-time. Oakland's Ada. Task_Indentification package provides the operations to read the string or the user may read the string via a debugger. Setting PJSP's Max_Task_Name_Length to zero suppresses the storage of agent names.

State Information The enumeration type Agent_State (Listing 42) defines the state of an agent or the state the agent wishes to enter. While extensive, most states target a specific primary agent and associate with the Oak Messages sent between kernel and agent. To simplify the kernel, the interrupted status of an agent lies outside the state component. An agent's state also records the its call stack, keeping tack of the location of a non-executing Agent's call stack together with its boundaries.

```
type Agent_State is
  (Bad_State,

   --  Activation States
   Activation_Pending, Activation_Failed,
   Activation_Successful, Activation_Complete,

   --  Main Execution States
   Running, Runnable, Sleeping, Terminated, Inside_PO,

   --  Waiting States
   Waiting_For_Event, Waiting_For_Protected_Object, Inactive,

   --  Interrupt Agent States
   Handling_Interrupt, Interrupt_Done,

   --  Request states
   Update_Task_Property,
   Setup_Cycles, New_Cycle, Raise_Cycle_Event, New_Cycle_Error,
   Entering_PO, Enter_PO_Refused, Exiting_PO, Exit_PO_Error,
   Attach_Interrupt_Handler, Waiting_For_Interrupt,
   Add_Agent, Remove_Agent,
   Set_Timing_Event_Handler, Cancel_Timing_Event,

   --  Scheduler Agent States
   Agent_State_Change, Selecting_Next_Agent, Wake_Agent,
   Adding_Agent, Adding_Agents, Removing_Agent,
   Scheduler_Agent_Done, No_Agent_To_Run,
   Initialising_Agents,
   Allowance_Exhausted, Not_Initialised,

   -- Internal States
   No_State, No_Message, Invalid_Message);
```

Listing 42
Oak.States
oak-states.ads

```
type Charge_Occurrence is
  (Only_While_Running, Same_Priority, All_Priorities,
   At_Or_Below_Priority);
```

Listing 43
Oak.Agent.Oak_Agent
oak-agent-oak_agent.ads

Scheduling Information Provides the basic information required to schedule the agent and records the scheduler agent responsible for scheduling it.

Execution Time Support Supports agent execution time tracking. It provides Oak the ability to support the *Cyclic Task Specification's* execution budgets and provides general timing information about an agent's execution to users.

Oak also facilitates the consumption of an agent's execution budget by other agents by placing the agent on the kernel's Budget_To_Charge list. To support Acton's implementation of execution servers, the When_To_Charge component (Listing 43) allows an agent on the Budget_To_Charge list to opt out from being charged based on the priority of the currently executing task.

Internal Oak Support Provides the record components to facilitate internal messaging and attachment to kernel lists.

kernel agent
Oak.Agent.Kernel

```
type Oak_Kernel_Record is record
   Current_Agent           : Oak_Agent_Id;
   Current_Priority        : Any_Priority;
   Current_Timer           : Oak_Timer_Id;

   Scheduler_Agent_Table   : Scheduler_Id_With_No;
   Interrupt_Agent_Table   : Interrupt_Ids;
   Interrupt_States        : Interrupt_Active_Set;

   Active_Protected_Brokers : Protected_Id_With_No;

   Budgets_To_Charge       : Charge_List_Head;
   Entry_Exit_Stamp        : Oak_Time.Time;

   Scheduler_Ops           : Scheduler_Ops_Stack;
end record;
```

Listing 44
Oak.Agent.Kernel
oak-agent-kernel.ads

A kernel agent represents a single instance of Oak. A non pre-emptive agent, a kernel agent controls access to a single processor and its memory resources. Amongst a kernel agent's responsibilities are dispatching agents, handling interrupts and timers, and managing agents and their requests. Requests a kernel acts on include: updating task properties, informing a task agent's scheduler agent of these changes if necessary; requests for a task to yield, sleep or start a new task cycle; controlling access to protected objects; and task activation.

The design of kernel agents entails the Oak_Kernel_Record records the kernel state that persists across invocations of the kernel. Storing the persistent state outside the kernel's stack allows easier access to the kernel data, removing the need to pass individual state variables or pointers between kernel subprograms.[8] Additionally, the kernel state becomes readable for debugging purposes and provides a fixed, convenient location to gather the current agent and priority of the system — used by the context switching subprograms and Oakland.

The persistent state records the currently dispatched agent, the current system priority and the next timer to fire. A kernel agent also details two tables of assistant agents: the *Scheduler Agent Table* lists the scheduler agents responsible for the system priority range; while the *Interrupt Agent Table* lists the interrupt agents responsible for handling interrupts, one per interrupt priority. As an internal optimisation, the kernel maintains a bit field indicating the active states of the interrupt agents, enabling the kernel to quickly identify active interrupt agents. In a similar role, the kernel tracks all active brokers.

Finally, Oak_Kernel_Record records kernel states associated with managing execution time tracking and pending scheduling operations. Of note, a kernel agent possess a list of agents who will have their execution budgets charged in response to the execution of current agent. The Budgets_To_Charge list enables an agent to consume another agent's execution budget in addition to its own, a feature used to implement execution servers.

8. A previous version of Oak took advantage of the persistent kernel state being stored outside the agent's stack by not saving the kernel's context during context switches. While reducing context-switching overhead, difficulties were encountered in implementing this approach in a stable manner across different processor architectures and optimisation levels.

scheduler agents
Oak.Agent.Schedulers

```
type Scheduler_Agent_Record is record
   -- Core scheduler agent components
   Lowest_Priority   : Any_Priority;
   Highest_Priority  : Any_Priority;
   Agent_To_Run      : Oak_Agent_Id;
   Run_Timer         : Oak_Timer_Id;

   -- Nested scheduler agents components
   Interpret_No_Agent_As : No_Agent_Interpretation;
   Charge_While_No_Agent : Boolean;
   Agent_Active_Till     : Active_Till;
   Period                : Oak_Time.Time_Span;
   Phase                 : Oak_Time.Time_Span;
   Relative_Deadline     : Oak_Time.Time_Span;
   Execution_Budget      : Oak_Time.Time_Span;

  Next_Run_Cycle     : Oak_Time.Time;
end record;
```

Listing 45
Oak.Agent.Schedulers
oak-agent-schedulers.ads

Scheduler agents provide Acton with scheduling services, scheduling other agents.[9] They manage a set of runnable queues; one for each priority level the agent is responsible for. Figure 21 presents the conceptual view of Acton's scheduling queues.

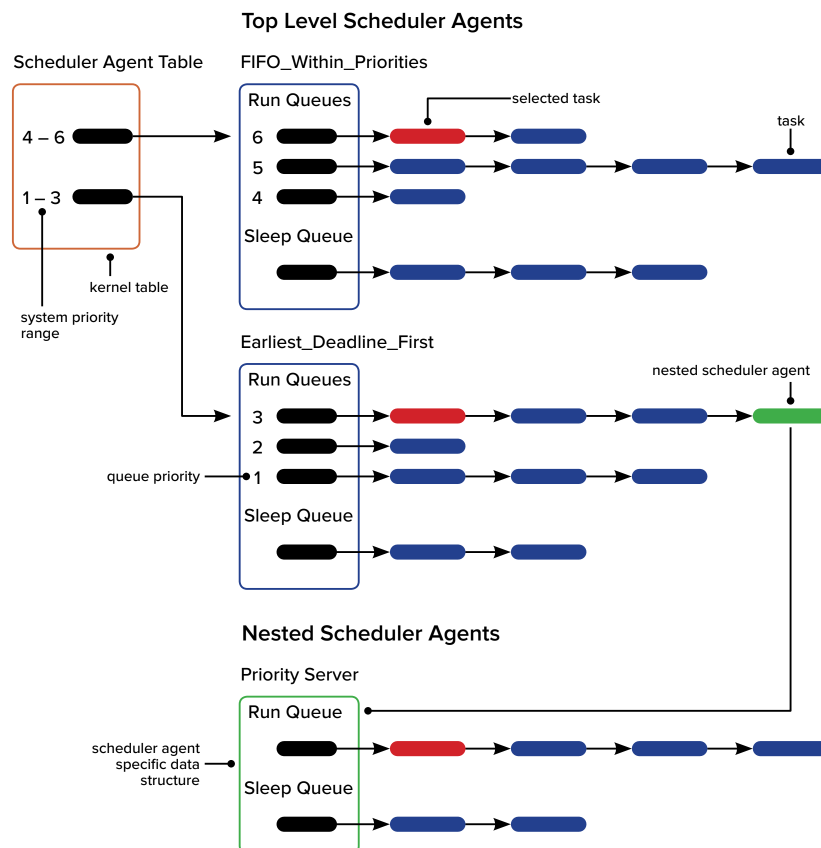


Figure 21
Conceptual view of Acton's scheduling arrangements.

9. Normally tasks and other scheduler agents, though nothing stops interrupt and kernel agents from being scheduled other than a lack of reason.

Oak dispatches a scheduler agent when the agent needs to manage its queues or when a kernel operation modifies the scheduling properties of an agent under its control. Oak communicates its reason for dispatching the scheduler agent via an Oak Message, with the scheduler agent providing a response in turn. The response informs Oak of the agent the scheduler agent wishes dispatched and the next time the scheduler agent needs to run again to manage its queues. While running, scheduler agents are non-pre-emptive to remove the possible contention caused when a pre-empting agent performs an operation requiring the interrupted scheduler agent; for example, when an interrupt handler releases a task blocked on an entry.[10]

Scheduler agents take on two roles within Acton. In their first role, scheduler agents mange the runnable queues making up the system defined priority range. Called top-level scheduler agents, Oak records them in the kernel's *Scheduler Agent Table* along with the priority range they manage. Oak uses the table to assign tasks a scheduler agent according to the task's priority if the task does not already specify a scheduler agent.

In their second role, scheduler agents may be nested: allowing a scheduler agent to schedule another. When a nested scheduler agent is selected for dispatch by its parent scheduler agent (as opposed to dispatch to perform queue management), Oak dispatches the agent nominated by the nested scheduler agent. Acton does not impose a limitation on the depth of scheduler agents nesting and technically allows the replacing and reassigning of top-level scheduler agents at run-time — though Acton does not currently use these abilities.

The scheduler agent record type (Scheduler_Agent_Record) separates into two sections to support Oak's interaction with the scheduler agents and to support nested scheduler agents.

Nested Scheduler Agents Oak's nested scheduler agents provides Acton's support for execution servers and is flexible to support servers like the Priority Server, Polling Server and Sporadic Server. To permit this range of execution servers, Oak offers a set of options to control the dispatching behaviour of a nested scheduler agent. These options relate to the management of the agent's active state and how Oak reacts to the selection of the No_Agent by the agent.

The active state of a nested scheduler agent determines whether the kernel will dispatch the scheduler's agents. While active, its parent schedules the nested scheduler agent if it has an agent to dispatch. When the nested scheduler agent deactivates, Oak removes it from its parent's queues.

Regardless of a nested scheduler agent's active state, Oak will directly dispatch it to perform scheduling operations; for example, adding and removing agents from the scheduler. An exception exists for inactive scheduler agents where queue management operations are deferred until the agent becomes active.[11]

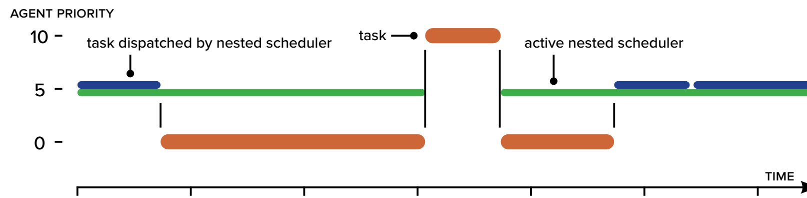
---

10. A main impediment to allowing pre-emptive scheduler agents is due to Oak queuing the scheduling operations and then dispatches these operations one at a time. If scheduler agents were pre-emptable, this queue would become unbounded in size.

11. Scheduler agents run so they can manage their queues in light of events occurring that will change the currently dispatched agent, like a task waking. Since an inactive scheduler agent cannot dispatch an agent, Oak safely delays its queue management until it becomes active.

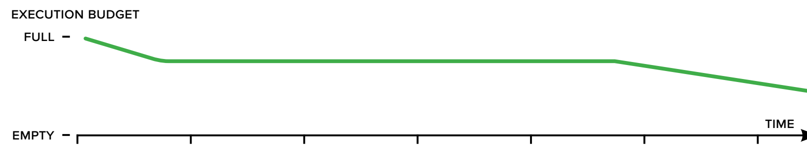
A nested scheduler agent dispatches tasks while inside a fixed-priority parent agent

Where the agent allows lower priority tasks to run when it has no tasks to dispatch



Its execution budget gets consumed depending on the agent's Charge_While_No_Agent property:

a) Charge_While_No_Agent = False



b) Charge_While_No_Agent = True

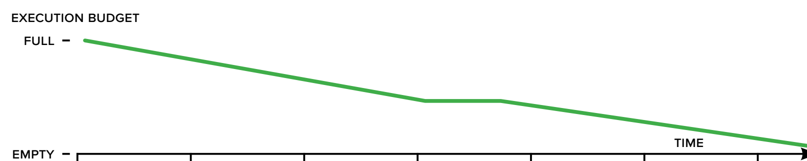


Figure 22
A schedule containing an active nested scheduler agent which allows lower priority tasks to run when it has no tasks to dispatch. The consumption of the scheduler's execution budget depends on the scheduler agent's Charge_While_No_Agent property:
(a) lower priority tasks do not consume the scheduler's budget;
(b) lower priority tasks do consume the scheduler's budget.

The agent state Allowance_Exhausted indicates the inactive state of a nested scheduler agent. Through the Agent_Active_Till component, Oak has the ability to automatically deactivate a nested scheduler once the nested scheduler agent's deadline has passed or after the exhaustion of its execution budget.

Similarly, Oak can periodically reactivate a nested scheduler agent, governed by the agent's period and phase. When the nested scheduler agent deactivates, Oak calculates the start of the next active period using the Next_Run_Cycle component and uses the agent's scheduler timer to signal when the nested scheduler can reactivate. Oak also replenishes the nested scheduler agent's execution budget and updates its deadline as appropriate using the values contained within the Scheduler_Agent_Record.

Consumption of a nested scheduler agent's execution budget occurs when agents from its queues execute. Agents outside scheduler can also consume the scheduler agent's budget if allowed via the Charge_While_No_Agent component. This option permits the preservation or consumption of the execution budget when the nested scheduler agent has no agents to dispatch (Figure 22).

A nested scheduler agent can also specify how Oak interprets its selection of the No_Agent as the agent to dispatch. While combining the No_Agent and sleep agent roles simplifies other aspects of Oak, nested scheduler agents present the sole case where differentiating between no agent selected and the sleep agent matters. Oak resolves the ambiguity through the Interpret_No_Agent_As component, allowing the nested scheduler agent to choose between dispatching the sleep agent (Figure 23 on page 108) or allowing lower priority agents to execute when the nested scheduler has no agents to dispatch (Figure 22).

The nested scheduler agent can also dispatch the Sleep Agent when it has no task to despatch.

Preventing the lower priority task from running until the scheduler deactivates.

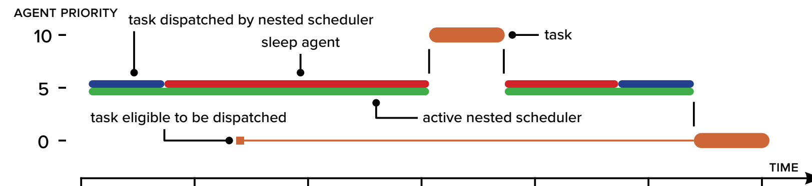


Figure 23 Nested scheduler agents dispatch the Sleep Agent when they have no agent to dispatch and Interpret_No_Agent_As equals Sleep_Agent. Figure 22 demonstrates when Interpret_No_Agent_As is As_Is.

Finally, Oak uses the Charge_While_No_Agent and Interpret_No_Agent_As components to set the nested scheduler agent's When_To_Charge component (part of the Oak_Agent type). Depending on the combination, the component will have a value of Same_Priority or At_Or_Below_Priority. When_To_Charge enables Oak to leave an active nested scheduler agent on the kernel's Budgets_To_Charge list while other agents run, simplifying Oak's implementation.

task agents

Oak.Agent.Tasks

```
type Task_Agent_Record is record
   Base_Priority : Any_Priority;

   --  Cyclic task properties
   Cycle_Behaviour      : Ada.Cyclic_Tasks.Behaviour;
   Cycle_Period         : Oak_Time.Time_Span;
   Phase                : Oak_Time.Time_Span;
   Execution_Budget     : Oak_Time.Time_Span;
   Relative_Deadline    : Oak_Time.Time_Span;
   Early_Event_Response : Ada.Cyclic_Tasks.Early_Event_Responses;

   Budget_Timer         : Oak_Timer_Id;
   Deadline_Timer       : Oak_Timer_Id;
   Next_Cycle           : Oak_Time.Time;
   Event_Raised         : Boolean;

   --  Protected object access properties
   Protected_Object : Protected_Id;
   Id_Of_Entry      : Entry_Index;
   Next_Queue_Item  : Oak_Agent_Id;

   --  Activation properties
   Activation_Link : Oak_Agent_Id;
   Elaborated      : Address;
end record;
```

Listing 46 Oak.Agent.Tasks *oak-agent-tasks.ads*

Task agents represent Ada tasks within Oak. The associated primary record type Task_Agent_Record provides support for the *Cyclic Task Specification*, Ada's task activation model and interactions with protected objects.

interrupt agents

Oak.Agent.Interrupts

Interrupt agents provide a context for interrupt and timer handlers to execute. Each interrupt priority has an interrupt agent responsible for all handlers assigned to that priority. This approach removes the need for tasks to reserve stack space for interrupt handlers. It also permits handlers to make kernel calls, which would cause problems if the handler ran in another agent's context.

```
   type Interrupt_Agent_Record is record
      Interrupt_Kind  : Interrupt_Type;
      External_Id     : External_Interrupt_Id;
      Timer_To_Handle : Oak_Timer_Id;
   end record;
```

Listing 47
Oak.Agent.Interrupts
oak-agent-interrupts.ads

Unlike task agents, scheduler agents do not schedule interrupt agents for efficiency reasons.[12] Instead, the kernel agent maintains a table of interrupt agents and dispatches an active interrupt agent over an agent specified by a scheduler agent if the interrupt agent has a equal or higher priority.

An interrupt agent supports two sources of interrupts: those external to the microcontroller core and those from the system timer. Oak provides the run-loop for interrupt agents, which looks up the Interrupt_Agent_Record object for the source of interrupt and executes the handler corresponding to the interrupt.

Brokers

Oak.Brokers

```
package Oak.Brokers is
   Protected_Id_Low_Bound  : constant := 1;
   Protected_Id_High_Bound : constant := Max_Protected_Agents;

   type Protected_Id_With_No is mod Protected_Id_High_Bound + 1;

   subtype Protected_Id is Protected_Id_With_No range
     Protected_Id_Low_Bound .. Protected_Id_High_Bound;

   No_Protected_Object : constant Protected_Id_With_No :=
                                       Protected_Id_With_No'First;
end Oak.Brokers;
```

Listing 48
Oak.Brokers
oak-brokers.ads

Brokers facilitate inter-task communication within Oak. Each type of broker has a specific broker record type, with a fixed storage pool and indexing system similar to Oak agents. At present, Oak only implements a single broker type: the protected broker representing protected objects.

Brokers originate from an earlier Oak implementation where protected agents, schedulable by scheduler agents, represented protected objects. The approach suffered from significant overhead adding and removing protected agents from scheduler agents as tasks entered and exited protected objects. Since then, protected agents turned into protected brokers, losing their schedulability.

protected brokers

Oak.Brokers.Protected_Objects

Oak controls access to Ada's protected objects and uses protected brokers to store the run-time state of protected objects. Besides recording the protected object's Oak_State (tracking whether the broker is active), the broker notes its ceiling priority and the address of the record object holding the protected object's contents. The broker also holds the protected object's entry queues and their barrier evaluation functions, together with a list of tasks active inside the object and wishing to gain access. The design of the protected broker enables Oak to support multiple entries per object with arbitrary length entry queues.

12. As a side effect, the delay statement is impotent for an interrupt agent; the kernel will simple ignore any request to sleep an interrupt agent. This does not causes any issue since an interrupt handler should never delay.

Listing 49
Oak.Brokers.Protected_Objects
oak-brokers-protected_objects.ads

```
type Protected_Broker_Record is record
   Name                    : Agent_Name;
   Name_Length             : Agent_Name_Length;

   Ceiling_Priority        : System.Any_Priority;
   Next_Object             : Protected_Id_With_No;
   State                   : Agent_State;
   Broker_Lock             : Boolean;

   Object_Record           : System.Address;
   Tasks_Within            : Oak.Agent.Agent_List;
   Contending_Tasks        : Oak.Agent.Agent_List;
   Entry_Queues            : Oak.Agent.Task_Id_With_No;
   Entry_Barriers          : System.Address;
end record;
```

To support brokers within the kernel, the protected broker provides a means to link brokers into lists. A relic of when protected brokers were initially conceived as agents, the broker also notes the name of the protected object the broker represents. Finally, their current design prohibits concurrent read access to protected objects since a protected broker can only be assigned to one kernel agent.

Oak Messages

Oak.Messages

Oak uses Oak Messages to facilitate communication between a kernel agent and other agents. Non-kernel agents use Oak Messages to request kernel services or to enter a new state. Kernel agents send Oak Messages to scheduler agents detailing the reason why Oak dispatched them. For now, other primary agents do not receive messages from the kernel as the kernel can reflect responses and requests by changing the agent's state.

Messages come in the form of the variant record Oak_Message (Listing 50), using Oak_States to specify requests and responses. Simplifying the messaging system, Oak considers a request for a kernel service (for example to attach an interrupt handler to a protected procedure) as a state change request, abet one where the requested state last for as long as the kernel service takes to complete.

Agents send Oak Messages to a kernel agent when they voluntarily context switch to the kernel as part of a system call, while a kernel agent sends Oak Messages as part of the agent dispatching process. Non-kernel agents' store the Oak Messages they send and receive on their stack, passing the location of the message store during the context switch to the kernel agent. The kernel agent then uses the location to exchange messages with the agent.

Listing 50
Oak.Messages
oak-messages.ads

```
type Oak_Message (Message_Type : Agent_State := No_Message) is
record
   case Message_Type is

      --  From Task Agents
      when Activation_Pending =>
         Activation_List : Task_List;

      when Sleeping =>
         Wake_Up_At              : Oak_Time.Time;
         Remove_From_Charge_List : Boolean;
```

Listing 50 continued
Oak.Messages
oak-messages.ads

```
      when Update_Task_Attribute =>
         Update_Task         : Task_Id;
         Attribute_To_Update : Task_Property;

      when Release_Task =>
         Task_To_Release : Task_Id;

      when Entering_PO =>
         PO_Enter        : Protected_Id;
         Entry_Id_Enter  : Indices.Entry_Index;

      when Exiting_PO =>
         PO_Exit : Protected_Id;

      when Attach_Interrupt_Handler =>
         Attach_Handler : Interrupt_Handler_Pair;

      when Add_Agent | Remove_Agent =>
         Agent     : Oak_Agent_Id;
         Scheduler : Scheduler_Agent_Id;

      when Set_Timing_Event_Handler =>
         Timer   : Oak_Timer_Id;
         Handler : Timing_Event_Handler;

      when Cancel_Timing_Event =>
         Timer_To_Cancel : Oak_Timer_Id;

      --  From Interrupt Agents
      when Interrupt_Done =>
         null;

      --  From Scheduler Agents
      when Scheduler_Agent_Done =>
         Next_Agent       : Oak_Agent_Id;
         Run_Scheduler_At : Oak_Time.Time;
         Run_Priority     : Any_Priority;

      -- To Scheduler Agents
      when Selecting_Next_Agent => null;

      when Adding_Agent | Adding_Agents =>
         Agents_To_Add : Oak_Agent_Id;
         Place_At      : Queue_End;

      when Removing_Agent | Agent_State_Change | Wake_Agent =>
         Target_Agent : Oak_Agent_Id;

      when Initialising_Agents =>
         Agents_To_Init : Oak_Agent_Id;

      when others => null;
   end case;
end record;
```

Oak Timers

Oak.Timers

```
type Oak_Timer_Kind is (Empty_Timer, Timing_Event_Timer,
                        Cycle_Timer, Scheduler_Timer);

type Oak_Timer (Kind : Oak_Timer_Kind := Empty_Timer) is record
   Fire_Time : Time;
   Priority  : Oak_Priority;

   case Kind is
      when Empty_Timer =>
        null;

      when Scheduler_Timer =>
         Scheduler : Scheduler_Id;

      when Cycle_Timer =>
         Timer_Action    : Ada.Cyclic_Tasks.Event_Response;
         Agent_To_Handle : Oak_Agent_Id;
         Handler         : Ada.Cyclic_Tasks.Response_Handler;

      when Timing_Event_Timer =>
         Event_Handler : Ada.Timing.Events.Timing_Event_Handler;
   end case;
end record;
```

Listing 51 Oak.Timers *oak-timers.ads*

Oak Timers provide the kernel data structures representing timers, associating a time with an event to be triggered by its passing. Storage for Oak Timers uses a similar technique to agents: with each timer allocated from a fixed storage pool referenced by an index type Oak_Timer_Id. Oak provides three forms of timers contained within a single mutable variant record: Oak_Timer. This approach differs from oak agent approach due to the simple design of the timers.

Oak Timers come in four forms: Cycle_Timer, Scheduler_Timer, Timing_Event_Timer and an Empty_Timer. A Cycle_Timer enforces a task agent's execution budgets and deadlines, while a Timing_Event_Timer supports Ada's timing events (ARM D.15). A Scheduler_Timer is used by Oak to support its scheduler agents: allowing Oak to dispatch the scheduler associated with the timer so the scheduler can service its queues, or activate and deactivate a nested scheduler agent as required. The Empty_Timer identifies an unused timer and the No_Timer. The No_Timer functions in the same manner as No_Agent: as an identifier indicating no selected timer.

A timer becomes active when added to the active timer queue, allowing the handling of the timer once its firing time has passed. However, Acton will delay handling a timer while the current priority of the system is higher than the timer's priority to prevent priority inversion.

key-priority red-back tree

All timers exist within an Oak Timer storage pool – implemented as an array in Oak. Active timers form an ordered set within the pool, sorted by increasing firing times. Oak implements the ordered set using a modified red-black tree called a key-priority red-black (KP-RB) tree, designed specifically for Oak. The KP-RB tree uses the same concepts as a normal red-black tree: a self-balancing binary search tree ordered by a key, offering $O(\log n)$ bounded insertion and deletion — where n is the number nodes in the set. The red-black tree offers

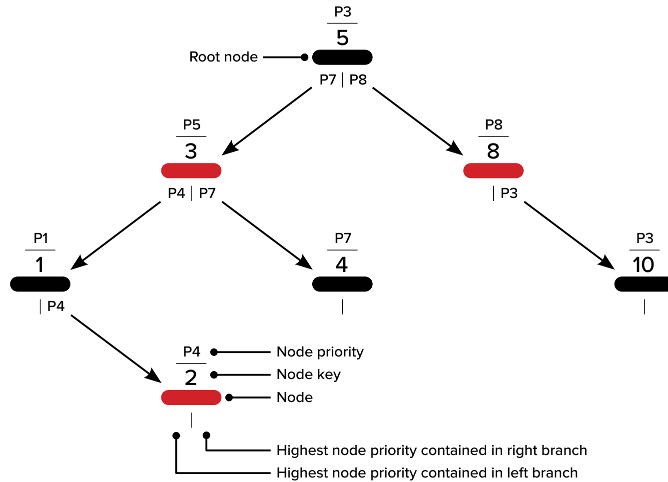


Figure 24
An example of a key-priority red-black tree.

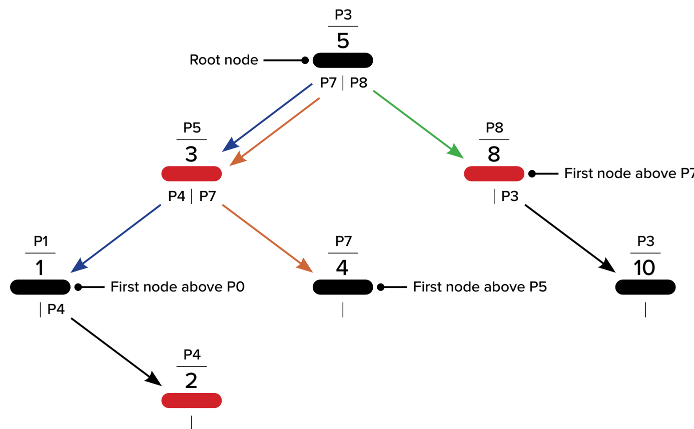


Figure 25
Searching for the first node with a priority above 0 (blue), 5 (orange) and 7 (green).

better insertion and removal complexity bounds than a linked-list's $O(n)$. For Oak, this is important as it removes and reorders unfired timers.

A KP-RB tree differs from a red-black tree by its ability to find the first item in the tree above a particular priority in a time bounded by $O(\log n)$. This unique ability enables Oak to efficiently look up the earliest firing timer above the system priority; ensuring a lower priority timer does not interrupt the running agent.

To implement the *first node above a priority* query, each node of the red-black tree records the maximum priority present in its left and right child branches (Figure 24). Maintenance of these values occurs as part of the red-black tree's normal node insertion and removal routines. The *first node above a priority* query search routine operates by descending the left branch of the tree until the routine reaches the first node of the tree or the maximum priority of the left branch becomes lower or equal to the lookup priority. For the latter, the routine will choose the current node if the node has sufficient priority or will repeat the search starting from the current node's right branch. Figure 25 demonstrates the search routine for several priorities.

The Oak Run-Loop

Oak.Core

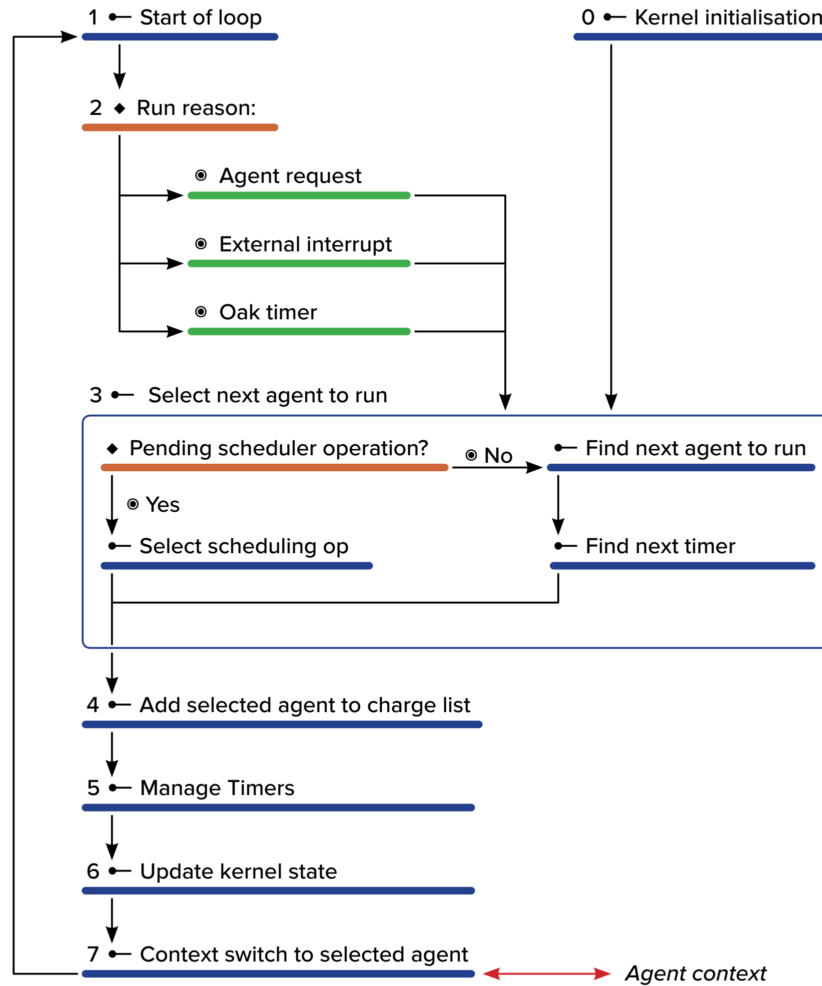


Figure 26
The Oak Run-Loop control flow.

The Oak Run-Loop — shown in Figure 26 — lies at the heart of Oak and is executed by its kernel agents. The run-loop implements the kernel's main logic: dispatching agents and responding to agent state requests. Additionally, the run-loop handles processor-raised interrupts, and manages agent execution time tracking and enforcement. A cycle of the run-loop commences when the processor's context switches to the kernel agent and completes when the kernel agent dispatches another agent. An iteration of the Oak Run-Loop executes non-preemptively to ensure consistency of the kernel's data structures.

1 — Start of Loop

A cycle of the Oak Run-Loop begins when the processor's context switches to the kernel agent. At this point, the kernel agent updates its execution time statistics, noting the time the kernel agent's current context began. The kernel agent also charges the execution time from the conclusion of the last Run-Loop cycle to the agents on the kernel's Budgets_To_Charge list (taking into account the When_To_Charge property for each agent on the list).

After updating the execution time statistics, the kernel agent records the reason the context switched to the kernel — the *reason for run* — and whether the

context switch was voluntary (as part of an agent service request) or because an interrupt was raised (from an external or timer interrupt). Furthermore, if an agent had voluntarily yielded the processor, the kernel agent records the location of the agent's Oak Message and copies the message into its stack.[13]

2 ◆ Run Reason

During the context switch to the kernel agent, the context switching routines provide a reason for the kernel agent running:

- An agent voluntarily yielded to Oak
- An external interrupt requires handling
- An Oak Timer requires handling

The latter two situations causes a running agent to involuntarily yield the processor to the kernel, which Oak notes in the agent's Agent_Interrupted property to enable correct restoration of the interrupted agent's context.

Once noted, this stage responds to the *reason for run*.

agent request

An agent voluntarily yields to the kernel agent to request a state change or an Oak provided services. As part of the yield, the agent provides an Oak Message containing the desired request (Listing 50 on page 110).

Here the kernel agent acts on the agent's request, with the agent ending in the requested state or passing through the requested state to another. For example, an agent requesting the Sleeping state will end up in that state if the supplied wake time lies in the future. Otherwise, the agent will remain in a Running or Runnable state, since the wake time has passed.

Handling an agent's request may result in the agent's scheduler requiring notification of changes to the agent's scheduling properties, or may see an agent added or removed from a scheduler agent. In these cases, the kernel agent pushes the required scheduling operations onto its Scheduler_Ops stack. Later in the run-loop, the kernel agent will dispatch these operations.

external interrupt

Here the kernel agent finds the interrupt agent responsible for handling the external interrupt by matching the priority of the interrupt to the corresponding agent located in the kernel agent's *Interrupt Agent Table*. It then transfers the external interrupt details to the interrupt agent's object and notes the interrupt agent as active within the kernel agent's Interrupt_States bit-field.

oak timer

Oak handles three kinds of Oak Timers: scheduler, cycle and timing event timers. The handling of a scheduler timer depends on the state of the timer's associated scheduler agent. If active and not time for a nested scheduler to deactivate, the kernel agent will push a Selecting_Next_Agent scheduler operation onto its Scheduler_Ops stack: allowing the scheduler agent to manage its queues. If it is time to deactivate a nested scheduler agent (since the agent exhausted its

13. Since agents pass the address of their Oak Message as part of the voluntary context switch.

budget or passed its deadline), the kernel agent removes the nested scheduler agent from its parent and arranges its next reactivation. Reactivation occurs when the nested scheduler agent's timer fires while it is inactive, causing the nested scheduler agent added back to its parent.

The handling of a cycle timer, on the other hand, depends on the Timer_Action property of the timer (which uses *Cyclic Task Specification's* Event_Response type). Since Oak only supports the No_Response and Handler responses, Oak treats the Abort_Cycle and Abort_Task options as No_Response, which the *Run-Loop* ignores. When responding with a handler, Oak follows a process similar to handling external interrupts: activating the corresponding interrupt agent and recording the handler to execute. Timing event timers use the same process. In all cases, the fired timer is deactivated.

3 •— Select Next Agent to Run

Once the run-loop has handled the *reason for run*, attention turns to selecting the next agent to dispatch. First, the kernel agent checks for scheduler operations on its Scheduler_Ops stack. If an operation is found, the kernel agent will pop the operation off its stack and select the associated scheduler agent as the next agent to run. The kernel agent copies the scheduling operation into the scheduler agent's Oak Message store and sets the priority of the system to maximum. The *Run-Loop* then progresses to the next stage.

If the Scheduler_Ops stack is empty, the kernel searches for the next agent to dispatch. First, the kernel agent consults its active interrupt agent and protected broker lists, choosing the agent or broker with the highest priority: selecting the broker if they have the same priority or the No_Agent if the kernel has no active interrupt agents or protected brokers. The kernel agent then selects the highest priority agent from its top-level schedulers if the agent has a priority greater than the currently selected agent or broker. The top-level scheduler query executes by reading the top-level scheduler agents' Agent_To_Run component.

At this point the kernel agent will have selected an agent or a broker, with the selected item's priority becoming the system priority. However, the kernel agent does not dispatch selected protected brokers or nested scheduler agents. Instead, if the kernel agent has selected a protected broker, it now elects to dispatch the task from the protected broker's Tasks_Within component: allowing the task inside a protected object to run. Similarly, if a nested scheduler agent has been selected, the kernel agent will select its Agent_To_Run in its place.

At this point, the kernel agent will have an agent to dispatch, even if it is the No_Agent (meaning no other agent was found). Since the No_Agent doubles as the sleep agent, the No_Agent is a valid selection: putting the processor to sleep when dispatched.

4 •— Add Agent to Charge List

A kernel agent maintains a *budgets to charge* list: a list of agents who will have their execution budgets consumed by the selected agent. At this step, the kernel agent adds the selected agent to the charge list so it consumes its own budget. This only occurs if the agent's execution budget is not Time_Span_Last.

5 •— Managing timers

The kernel agent finds the first firing timer and sets the system timer accordingly provided the kernel has not selected a scheduler agent (since scheduler agents are non-preemptable). To find the timer, the kernel agent searches the active timer queue to find the first timer above the selected system priority. Since timers associated with execution budget enforcement are not kept in the active timer queue, the kernal agent then selects the agent on its Budget_To_Chage list with the smallest remaining execution budget. This selected agent has its associated budget timer updated with a time equal to the current time plus the remaining execution budget. The first firing timer is then selected from the selected budget and active timers.

6 •— Update kernel state

The kernel agent now updates the system state before dispatching the selected task, recording the selected agent and timer within its Oak_Kernel_Record for reference outside the kernel agent's context. The kernel agent also adjusts the hardware priority level to match the selected system priority if the external interrupt controller supports multiple interrupt priority levels, preventing a lower priority interrupts from interrupting an executing agent.[14]

Finally, the kernel agent updates its timing statistics. The kernel agent charges the time spent executing the current iteration of the run-loop to itself and records the kernel's exit time so the kernel agent can determine the execution time of the dispatched agent when context returns to the kernel.

7 •— Context Switch to Agent

The final stage of the *Run-Loop* dispatches the selected agent via a context switch to the agent. As an optimisation, the kernel agent will skip the context switch if the selected timer has already fired, commencing a new iteration of the run-loop to handle the fired timer.

Two context switching routines exist to dispatch the agent. If the dispatching agent's was previously interrupted, Oak restores the agent's full context. If not, the kernel only restores only the caller-saved registers since the agent yielding routines already save the callee-saved registers.

0 •— Kernel Initialisation

A kernel agent starts the *Oak Run-Loop* for the first time during its initialisation. As its final initialisation step, the kernel agent pushes two scheduling operations onto its Scheduler_Ops stack: one allowing the top-level scheduler agents to initialise themselves and another to add the main environment task to its scheduler agent. Consequently, the kernel agent starts the run-loop at *3 – Select Next Agent to Run* because it has scheduling operations to dispatch.

14. Oak supports multiple hardware interrupt priorities by mapping them to Ada's interrupt priority range Interrupt_Priority.

Scheduling

Task scheduling occurs outside Oak, with scheduler agents preforming the scheduling role. Each scheduler agent has responsibility over a set of runnable queues, mappable to a system priority range. A scheduler agent also maintains a queue of agents under its control who are sleeping. The objective of a scheduler agent lies in managing its queues and nominating an agent to dispatch.

Two forms of scheduler agents exist: top-level and nested. A top-level scheduler agent has responsibility for all or part of the system priority range defined by Ada while a nested scheduler agent acts like a task, assigned to another scheduler agent for scheduling. In the latter form, the nested scheduler agent will have its nominated agents dispatched only when its parent selects it. Increasing their flexibility, Oak can automatically activate and deactivate nested scheduler agents to allow them to be used as execution servers.

Oak's limited role in scheduling consists of managing scheduler agents: dispatching them when they need to manage their queues, informing them of agent state changes, and adding and removing agents from their control. Additionally, Oak maintains its top-level scheduler table and supervises nested scheduler agents: deactivation them as their execution budgets and deadlines expire, and tending to their periodic reactivation.

By separating scheduling from task dispatching and removing it from the kernel, Acton can supports different types of schedulers; including different schedulers for different priority ranges. Development of these schedulers can also occurs independently of the kernel. This section examines the operation of scheduler agents and how Oak provides their support.

Scheduler Agent Operation

Scheduler agents determine their own implementation, constrained by the following restrictions:

- Cannot call on any kernel services or request a state change;
- Shall perform the action requested by the kernel;
- Shall provide a Scheduler_Agent_Done message when yielding with an agent to dispatch and a time when it wishes to run again; and
- Can read an agent's object stored in the kernel, but cannot modify it.

A scheduler is free to pick a data structure for its queues that suits its objectives. For example, the provided FIFO_Within_Priorities scheduler uses a single priority queue for its runnable queues and a KP-RB tree for its sleep queue. By contrast, the provided Priority_Server scheduler uses a binary heap for both queues since it manages a single priority.

In light of the restrictions, a scheduler will generally implement its scheduling operations within the run-loop shown in Figure 27, storing supporting data structures on its stack. Since a scheduler agent cannot modify an agent's kernel objects, a scheduler needs to populate its queues with a data structure representing the agent and any additional data it requires.

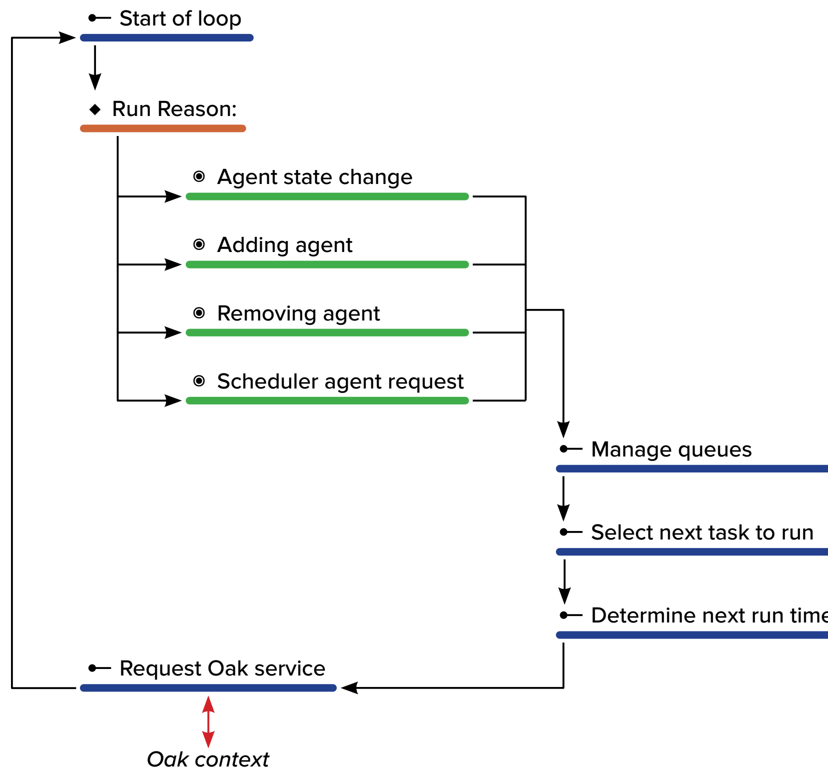


Figure 27
A scheduler agent run-loop.

The run-loop has single entry-exit point where the scheduler agent yields to the kernel once it has completed a scheduling operation. Later, when Oak dispatches the scheduler agent to perform another operation, the scheduler agent resumes from this point. Oak provides the requested scheduling operation as an Oak Message and can request one of five operations:

Add an agent (Adding_Agent) Adds an agent to the scheduler agent for scheduling. The agent will be added to the runnable queues if the agent's wake time has passed; otherwise, the agent ends up on the sleep queue.

Remove an agent (Removing_Agent) Removes the specified agent from the scheduler agent.

Wake agent (Wake_Agent) Moves the specified agent from the sleep queue to the runnable queue associated with the agent's prioirty.

Notify scheduler agent (Agent_State_Change) Informs the scheduler agent that the specified agent's state has changed, allowing the scheduler to organise its queues in response. For example, if a task enters the Sleeping state, the scheduler moves the task to the sleep queue. Oak notifies a scheduler about the following agent state changes:

- Wake time
- Absolute deadline
- Priority
- Entering the Sleeping state

Service Scheduler Agent (Selecting_Next_Agent) Allows a scheduler agent to service its queues. Oak requests this operation in response to the scheduler agent's timer firing. For example, the operation enables a scheduler agent to move woken agents from the sleep queue to their runnable queues and allows a Round-Robin scheduler to implement time slices. Since a scheduler agent manages its queues in the next step of the run-loop, the agent can elect to do nothing here in response to the Selecting_Next_Agent request.

Following the servicing of the kernel's request, the scheduler agent can manage its queues. The run-loop then completes with the scheduler agent yielding to the kernel with the Scheduler_Agent_Done Oak Message. This message contains the agent to dispatch, the time the scheduler wishes to execute again and the priority its scheduler timer. The scheduler agent will nominate No_Agent if it has no agent to dispatch, while the priority parameter allows Oak to not interrupt a running agent if the scheduler agent is only going to manage lower priority agents.

Implemented Scheduler Agents

Acton currently supplies two scheduler agents: a FIFO_Within_Priorities scheduler (Acton.Scheduler_Agents.FIFO_Within_Priorities) and a Priority Server (Acton.Scheduler_Agents.Priority_Server).

The FIFO_Within_Priorities scheduler uses a priority queue for its runnable queues, implemented as a red-black tree. A KP-RB tree provides the sleep queue to enable the scheduler to efficiently pick the first waking agent with a priority above the agent selected for dispatch; ensuring the scheduler does not interrupt the selected agent just to tend to lower priority tasks.

By contrast, the Priority_Server scheduler uses a binary heap for its queues because, as an execution server, it only looks after a single runnable queue. The binary heap provides an efficient data structure for storing and ordering agent IDs because only agent IDs need storing in the underlying array and queue operations are $O(\log n)$ bounded.

Management of Scheduler Agents by Oak

Oak.Scheduler

Oak manages scheduler agents as part of its run-loop. The kernel's interaction occurs in four places: handling agent requests and scheduler timers (*Oak Run-Loop: 2 – Run Reason*), and dispatching scheduling operations and finding the highest priority agent to dispatch (*Oak Run-Loop: 3 – Select Next Agent to Run*).

While handling agent requests, a kernel agent places the scheduling operations it requires on its Scheduler_Ops stack. Once the kernel agent completes the agent request, it performs the scheduling operations by dispatching the corresponding scheduler agent with the requested operation. This arrangement enables the *Oak Run Loop* to have a single agent dispatching point, simplifying the kernel.

The *Oak Run-Loop* directly handles the dispatching of scheduler operations with the remaining management operations provided by the Oak.Scheduler package (Listing 52).

Listing 52
Oak.Scheduler
oak-scheduler.ads

```
package Oak.Scheduler with Preelaborate is

   --  Scheduler Table Queries

   procedure Check_Sechduler_Agents_For_Next_Agent_To_Run
     (Next_Agent_To_Run : out Oak_Agent_Id;
      Top_Priority      : out Any_Priority);

   function Find_Scheduler_For_System_Priority
     (Priority : Any_Priority;
      CPU      : System.Multiprocessors.CPU_Range)
      return Scheduler_Id_With_No;

   --  Scheduling Operations

   procedure Add_Agent_To_Scheduler
     (Agent    : in Oak_Agent_Id;
      Place_At : in Queue_End := Back);

   procedure Add_Agents_To_Scheduler (Agents : in Oak_Agent_Id);

   procedure Inform_Scheduler_Agent_Has_Changed_State
     (Changed_Agent : in Oak_Agent_Id);

   procedure Remove_Agent_From_Scheduler
     (Agent : in Oak_Agent_Id);

   --  Scheduler Agent Management

   procedure New_Scheduler_Cycle (Scheduler : in Scheduler_Id);

   procedure Post_Run_Scheduler_Agent
     (Agent   : in Scheduler_Id;
      Message : in Oak_Message);

   procedure Service_Scheduler_Agent_Timer
     (Scheduler : in Scheduler_Id);

end Oak.Scheduler;
```

The Oak.Scheduler package provides three kinds of management operations:

- Queries on the scheduler table;
- Sending scheduling operations to scheduler agents; and
- Handling a scheduler agent's return message or the firing of its timer.

scheduler table queries

The Oak.Scheduler package supplies two queries on the kernel agent's top-level scheduler table. The first query, Find_Scheduler_For_System_Priority, returns the top-level scheduler agent responsible for the specified system priority. Oak uses the query when adding a new task to determine which scheduler agent has the responsibility for scheduling the agent.

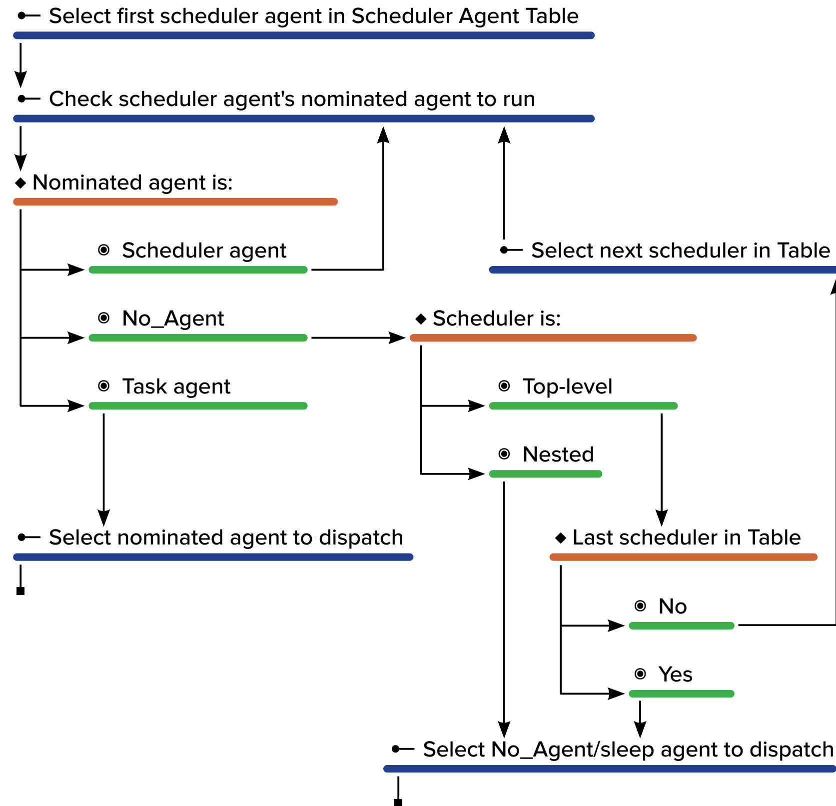


Figure 28
Searching for the next agent to run via the Check_Sechduler_Agents_For_Next_Agent_To_Run procedure.

The second query, Check_Sechduler_Agents_For_Next_Agent_To_Run, searches the top-level scheduler table for the highest priority agent to dispatch, with Figure 28 describing its operation. Oak uses the priority returned by the procedure as the effective priority of the selected agent when setting the system priority to mask lower priority external interrupts and timers.

scheduling operations

The Oak.Scheduler package provides a set of procedures to push scheduling operations onto a kernel agent's Scheduler_Ops stack. These procedures provide an abstraction over the stack operations to make the operations more readable.

Note in the case of the Add_Agents_To_Scheduler operation, scheduler agents do not perform this operation because the agents on the list may belonging to different scheduler agents. Instead of splitting the list into sub-lists, Oak handles the operation by adding the set of agents to their scheduler agents one at a time.

scheduler agent management

The main management procedures handle the two scheduler-related reasons for a kernel agent to run: a scheduler timer fired or a scheduler agent yielded. Handling the timer is the Service_Scheduler_Agent_Timer procedure, while the Post_Run_Scheduler_Agent procedure responds to a scheduler agent yield.

Service_Scheduler_Agent_Timer When a scheduler agent's timer fires, the event signals three possibilities: the scheduler agent requires dispatching to services its queues, or the time has come for a nested scheduler agent to become active or inactive.

For a scheduler agent wanting its queues serviced or a nested scheduler agent being reactivated, the procedure simply pushes a Selecting_Next_Agent operation onto the Scheduler_Ops stack. For a deactivating nested scheduler agent, the procedure calls New_Scheduler_Cycle: removing the nested scheduler agent from its parent; refreshing its execution budget and deadline; and setting the agent to reactivate at the start of its next active period by setting the scheduler agent's timer to this time.

Post_Run_Scheduler_Agent At its simplest, the procedure takes the Scheduler_Agent_Done message received from the scheduler agent and updates the agent's kernel object. Specifically, the scheduler agent's Agent_To_Run component is set to the selected agent, while its timer updates to the time the scheduler wants to run again.

Increasing its complexity, Post_Run_Scheduler_Agent also handles the intricacies of nested scheduler agents. This involves managing the two scheduler timer roles (running the scheduler agent and deactivation), the nested scheduler agent's presence on scheduler queues and its reactivation.

Figure 29 on page 124 presents the procedure's control flow. In detail:

1 •— The scheduler agent's Agent_To_Run component is set to the agent nominated in the Scheduler_Agent_Done message.

2 •— The nominated agent's state is set to runnable. Oak performs the operation because the scheduler agent cannot modify the agent's state.[15]

3 •— If a top-level scheduler agent the procedure completes after the scheduler agent has its timer updated to the Run_Scheduler_At and Run_Priority components of the Scheduler_Agent_Done message.

4 ◆ Otherwise, the procedure handles the nested scheduler agent. If an active nested scheduler agent, the procedure places it on the kernel agent's Budgets_to_Charge list, unless the scheduler has nominated the No_Agent and does not allow the No_Agent to consume its execution budget. Oak requires this step as the kernel always removes the last dispatched agent from the Budgets_to_Charge list at the start of its run-loop.

5 ◆ The nested scheduler agent's scheduler timer updates to reflect the next time the scheduler needs servicing. The action depends on the active state of the scheduler agent and whether it deactivates on execution budget exhaustion or once a deadline has passed:

- If active and will deactivate at a deadline sooner than the requested run time, the deadline becomes the firing time of the scheduler timer. Otherwise, while the scheduler is active, the timer uses the requested run time.

- If inactive, the timer sets to the time of reactivation.

15. As a side effect, woken tasks on runnable queues keep their Sleeping state until selected. This does not affect Oak, since no kernel operation depends on runnable tasks having a Runnable state.

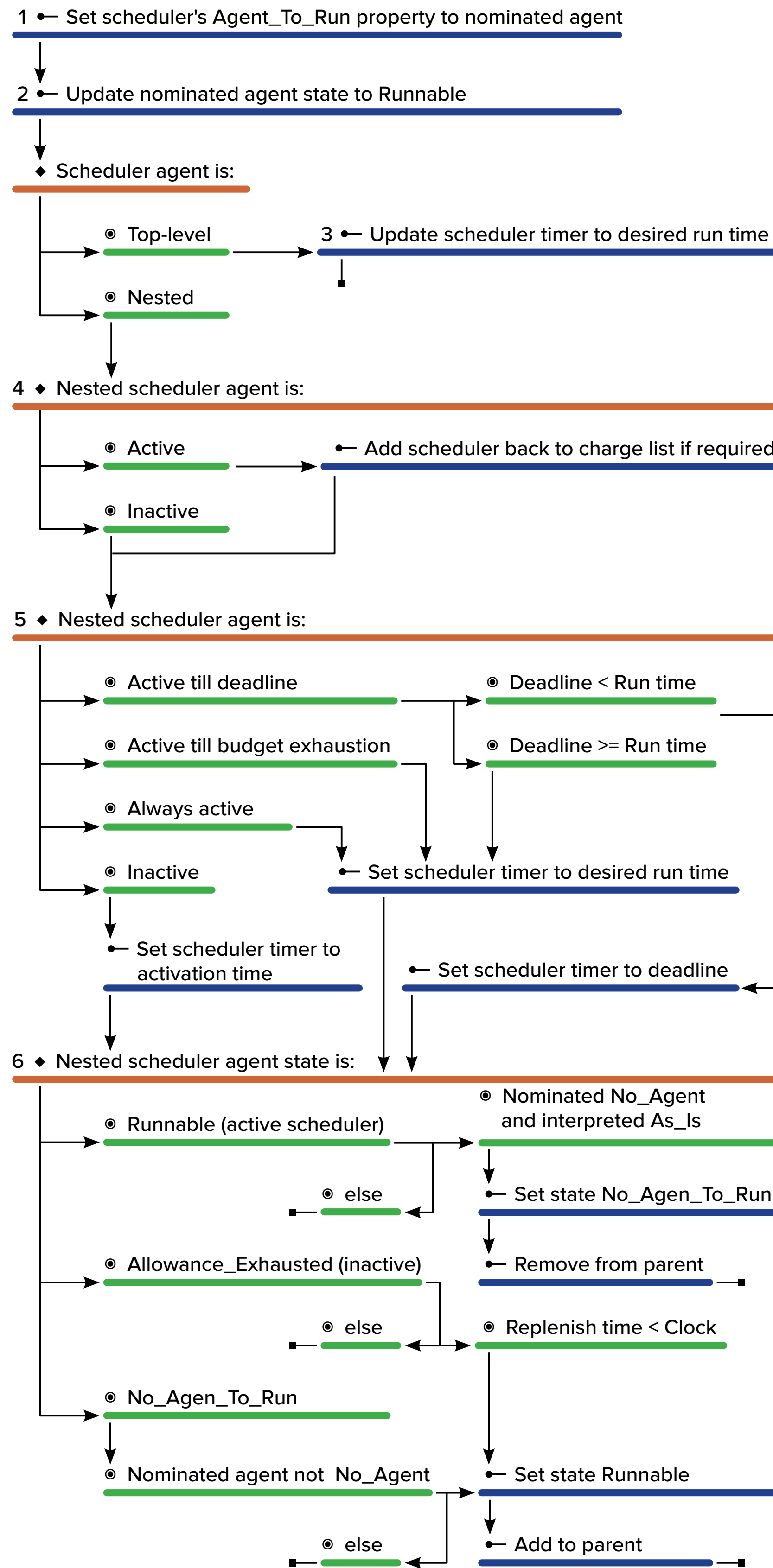


Figure 29
Control flow for Oak.Scheduler.Post_Run_Scheduler_Agent.

6 • The final step handles the reactivation of an inactive scheduler agent and the case when an active scheduler agent allows lower priority agents to execute when it has no agents to dispatch. To reactivate a scheduler agent, the procedure updates its state to Runnable and adds the agent back to its parent providing it has an agent to dispatch (including the sleep agent). If a scheduler agent allows lower priority agents to run in its place, the scheduler agent is removed from its parent while it has no agents to dispatch. Oak removes scheduler agents in this instance to simplify their implementation so they do not have to treat scheduler agent differently from task agents.

Task Management

Acton provides an implementation of Ada's task semantics (ARM 9) directly in Oak. Oak combines the kernel's task data structures (*Oak Agent* on page 102 and *Task Agent* on page 108) and scheduling (*Scheduling* on page 118) with task management operations to implement the required tasking semantics. Oak currently provides three task management operations: creation, activation and delay.[16] Ada's tasking syntax provides access to these task management routines, which Acton exposes to the compiler through its run-time library Oakland in a series of wrapper procedures around the associated kernel calls.

Task Creation

Oak.Agent.Tasks

Acton creates a new task with the New_Task_Agent procedure, which returns with the Task_ID of the new task. The procedure allocates to the new task an Oak agent object and a task agent object from their respective storage pools, initialising these objects with the parameters provided to the procedure:

```
procedure New_Task_Agent
  (Agent              : out Task_Id;
   Stack_Address      : in Address;
   Stack_Size         : in Storage_Count;
   Name               : in String;
   Run_Loop           : in Address;
   Task_Value_Record  : in Address;
   Base_Priority      : in Integer;
   Cycle_Behaviour    : in Ada.Cyclic_Tasks.Behaviour;
   Cycle_Period       : in Oak_Time.Time_Span;
   Phase              : in Oak_Time.Time_Span;
   Execution_Budget   : in Oak_Time.Time_Span;
   Budget_Response    : in Ada.Cyclic_Tasks.Event_Response;
   Budget_Handler     : in Ada.Cyclic_Tasks.Response_Handler;
   Relative_Deadline  : in Oak_Time.Time_Span;
   Deadline_Response  : in Ada.Cyclic_Tasks.Event_Response;
   Deadline_Handler   : in Ada.Cyclic_Tasks.Response_Handler;
   Scheduler_Agent    : in Scheduler_Id_With_No := No_Agent;
   Chain              : in out Task_List;
   Elaborated         : in Address := Null_Address);
```

Listing 53
New_Task_Agent defintion
Oak.Agent.Tasks
oak-agent.tasks.ads

16. Acton does not provide task finalization and termination, following the Ravenscar Profile.

Most of the procedure's parameters map to the objects' components, allowing the procedure to store the values passed. As an exception, if the specified scheduler agent is No_Agent the procedure assigns the top-level scheduler agent responsible for the task's priority. The procedure also permits the caller to provide a pre-allocated stack via the Stack_Address parameter or let Oak perform the allocation itself by specifying Null_Address. The Stack_Size parameter indicates the size of the provided stack or the stack size Oak will allocate.

Once the task agent has a stack, the procedure initialises the stack with the task's initial context. This includes setting the task's instruction pointer to the procedure representing the task body and storing the address of the task's Task_Value_Record (the record object created by GNAT containing the task's state).

As its last step, the procedure adds the new task agent to the activation list provided by the Chain parameter.

To support Acton's new task create call, GNAT for Acton modifies the Make_Task_Create_Call procedure within GNAT's EXP_CH9 package.

Task Activation

Oak.Agent.Tasks.Activation

Ada defines the initial phase of a task's execution as its activation: elaborating the declarative part of the task's body (ARM 9.2). Furthermore, all tasks created by the elaboration of object declarations in a declarative region (including sub-objects) activate together.[17] Failure due to the propagation of an exception during activation causes the agent to become completed and sees Tasking_Error raised in the task that initiated the activation: the activator. The activator blocks until all task activations complete before progressing.

The GNAT compiler provides task activation support as part of its expansion of Ada tasks, inserting GNARL run-time calls in the user code to implement the run-time semantics. Acton replaces these calls with equivalent Oakland calls.

To activate tasks declared in a declarative region, GNAT creates an activation chain object (_chain) when the compiler encounters the first task object declaration. GNAT then adds every task declared within that declarative region to the chain via the task creation call Oak.Agents.Task_Agents.New_Task_Agent. Internally, the chain consists of a linked-list, linked through the task agent's Activating_List component as illustrated by the top of Figure 30.

At the end of the declarative region, GNAT inserts a run-time call to the Oakland.Tasks.Activate_Tasks procedure with the populated activation chain: a call which commences the activation of the tasks on the chain as described in Figure 31 on page 129. The activation processes consists of six stages:

1 •— Oakland.Tasks.Activate_Tasks takes the activation chain and ensures the bodies of the tasks on the chain have elaborated, raising Program_Error otherwise. The procedure then yields to the kernel with an Activation_Pending message containing the activation chain. Once task activation completes, the context returns and the procedure checks the activator's state to determine whether the activation was successful. If so the procedure returns, otherwise the procedure raises Tasking_Error.

17. RM 9.2 also states tasks created from a single allocator activate together, however Oak does not support allocators.

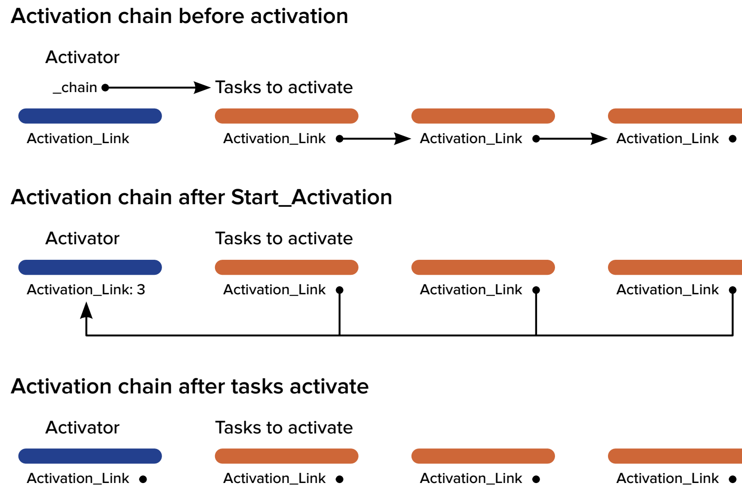


Figure 30
Progression of an activation chain before, during and after the activation of tasks on the chain.

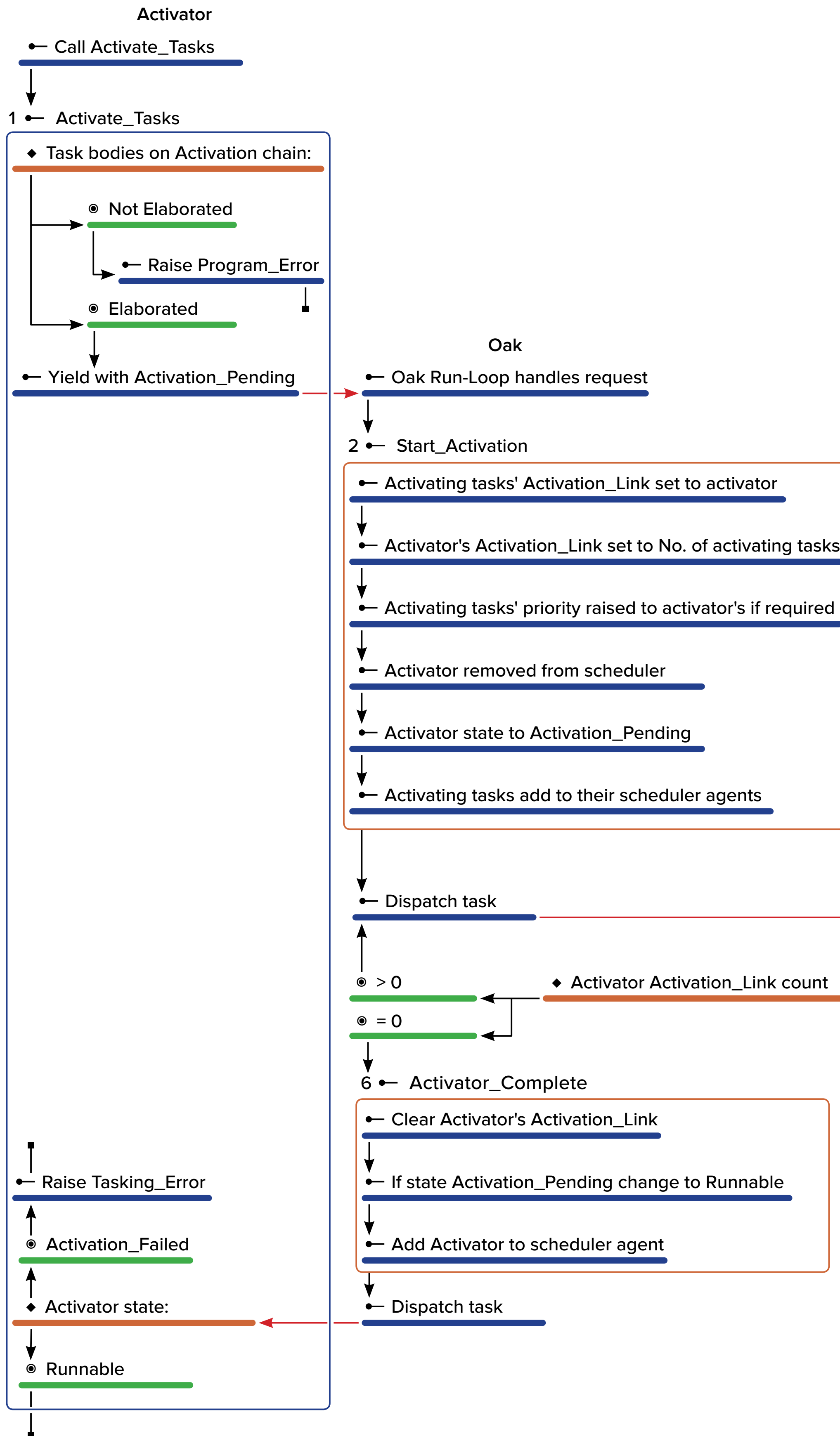
2 The Oak.Agent.Tasks.Activation.Start_Activation procedure handles the received Activation_Pending message within Oak. As shown in the middle diagram of Figure 30, the procedure restructures the activation chain: each activating task's Activation_Link now points to the activator, while the activator's Activation_Link contains the number of activating tasks. These changes enable Oak to separately dispatch activating tasks and identify when the chain's activation has completed.

Oak does not dispatch the activator until activation completes, causing the procedure to remove the activator from its scheduler agent and updating its state to Activation_Pending. Start_Activation then proceeds to add the activating tasks to their scheduler agents, temporarily raising a task's priority to the activator's priority if the task is below it, as per ARM D.1.21. At this point, the activating tasks are schedulable and dispatched by Oak when nominated by their scheduler agent.

3 When dispatched, an activating task will elaborate its declarative statements. If successful, the task calls Oakland.Tasks.Complete_Activation: yielding to the kernel with an Activation_Successful message. Otherwise, the raising of an exception within the declarative statements causes the task to complete by calling Oakland.Tasks.Complete_Task. The Complete_Task procedure discerns normal task completion from failed activation by the task's Activation_Link: if the link points to another task then activation has failed and the task yields to the kernel with an Activation_Failed message.

4 The Oak.Agent.Tasks.Activation.Activation_Complete procedure handles the Activation_Successful message within Oak. The associated task has its Activation_Link cleared (Figure 30) and priority restored if previously raised. The activator sees its Activation_Link count decremented, with Oak.Agent.Tasks.Activation.Activator_Complete called when the counter reaches zero.

5 Oak.Agent.Tasks.Activation.Activation_Failed, by contrast, handles the Activation_Failed message within Oak by removing the failed task from its scheduler agent and setting its state to Terminated. The activator has its state set to Activation_Failed and Activation_Link decremented, with the procedure calling Activator_Complete when the counter reaches zero.


Activator
Call Activate_Tasks
1 Activate_Tasks
Task bodies on Activation chain:
Not Elaborated
Raise Program_Error
Elaborated
Yield with Activation_Pending
Oak
Oak Run-Loop handles request
2 Start_Activation
Activating tasks' Activation_Link set to activator
Activator's Activation_Link set to No. of activating tasks
Activating tasks' priority raised to activator's if required
Activator removed from scheduler
Activator state to Activation_Pending
Activating tasks add to their scheduler agents
Dispatch task
> 0
Activator Activation_Link count
= 0
6 Activator_Complete
Clear Activator's Activation_Link
If state Activation_Pending change to Runnable
Add Activator to scheduler agent
Dispatch task
Raise Tasking_Error
Activation_Failed
Activator state:
Runnable

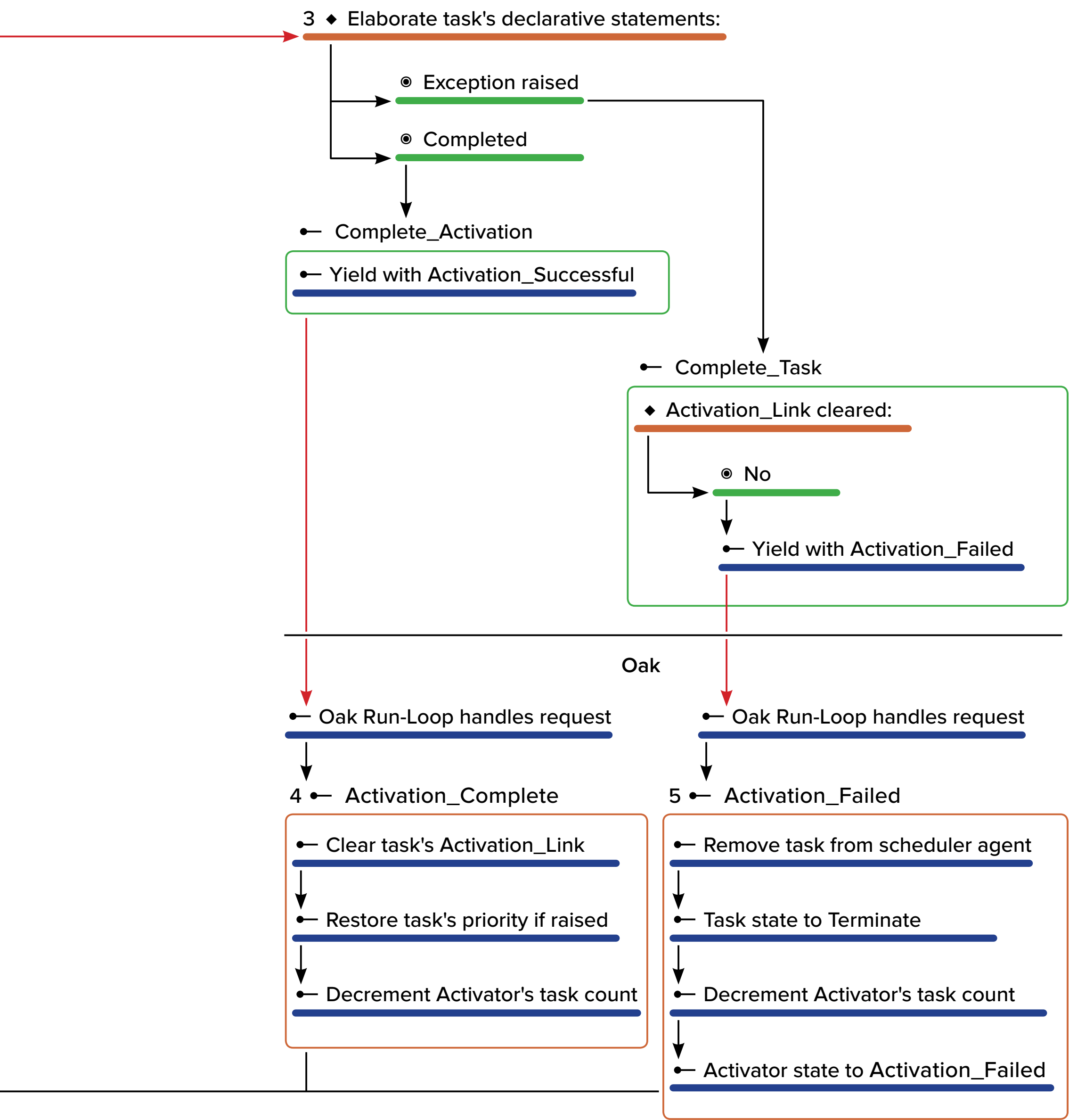


Figure 31
Task activation

6 ⟵ The Oak.Agent.Tasks.Activation.Activator_Complete procedure clears the activator's Activation_Link and updates the state of the activator to Runnable if its state is not Activation_Failed. Finally, the procedure adds the activator back to its scheduler agent, allowing the activator to resume and complete its Oakland.Tasks.Activate_Tasks procedure.

Task Delay

Ada.Real_Time.Delays

With a focus on real-time systems, Acton only supports Ada's monotonic time. Ergo, the only delay statement Acton provides is the monotonic time delay until statement. GNAT provides its implementation through the Ada.Real_Time.Delays.Delay_Until procedure, which Acton replaces with its own. Acton's implementation forms a lightweight wrapper around a sleep request to Oak using the Sleeping Oak Message. Oak handles the message by updating the task agent's state to Sleeping, setting the agent's Wake_Time component to the requested wake time and informs the task's scheduler agent of the changes made.

Cyclic Tasks

Acton supports most of the *Cyclic Task Specification*, missing support for the Abort_Cycle and Abort_Task event responses and the Ada.Execution_Servers.User_Servers package. Oakland provides the library packages documented by the *Specification* and the run-time routines inserted by the GNAT compiler front-end. In turn, Oak incorporates the new task attributes into the Oak_Agent and Task_Agent record types: Oak_Agent housing components related to execution time tracking while Task_Agent provides support for the other cyclic task attributes. Oak also implements the dynamics of the *Specification*: centred around Oak Messages received from Oakland and the timers associated with the task agent objects. Finally, Acton uses the *Cyclic Task Specification's* execution servers as a public interface to its nested scheduler agents.

Oakland Support

Oakland.Tasks

Oakland provides the two parent packages — Ada.Cyclic_Tasks and Ada.Execution_Servers — and the two corresponding Dynamics child packages defined by the *Cyclic Task Specification*. Most of the subprograms these packages provide retrieve and set the task attributes defined by the *Specification*, making their implementation straightforward. Attribute retrieval functions access their respective task attributes directly, while the set procedures rely on Oak to perform the update on their behalf: with the kernel informed via an Update_Task_Property message with the attribute to modify and its new value.

The few package procedures not simply updating a task attribute include the Add_Task_To_Server and Remove_Task_From_Server procedures within the Ada.Execution_Servers package. These procedures instead use the corresponding Oak Messages.

Separate from the *Specification's* packages, the Oakland.Tasks package provides the internal procedures used by the compiler during the expansion of the cyclic task bodies and the **new cycle** statement: Begin_Cyclic_Stage, New_Cycle and Raise_Cycle_Event. A task body calls Begin_Cyclic_Stage just before the start of its cyclic section, while New_Cycle signals the end of one cycle and the start of another. Similarly, the Raise_Cycle_Event procedure raises a cycle event for the specified sporadic or aperiodic tasks. Lightweight wrappers around kernel calls implement the three procedures using corresponding Oak Messages.

Oak Support

Oak's agent and timer objects incorporate the necessary data structures to support the *Cyclic Task Specification* (as described in *Oak Resources* on page 98). The Oak_Agent type holds the Absolute_Deadline and execution time related task attributes because they have a broader use outside of cyclic tasks: the Absolute_Deadline attribute enables the scheduling of any agent under a deadline-based scheduler; while the execution time attributes allows all agents to have their execution time tracked. The Task_Agent type houses the remaining attributes due to their task specific nature.

Oak implements the *Specification's* dynamic semantics as reactions to Oak Messages from cyclic tasks and to cycle timers firing. Six related requests exist:

- Setup_Cycles
- New_Cycle
- Commence_New_Cycle
- Update_Task_Properties
- Add_Agent
- Remove_Agent

The last two kernel requests support execution servers: adding and removing execution servers from the system in addition to adding and removing tasks from execution servers. These operations work on both execution servers and tasks because nested scheduler agents underlie Acton's execution server implementation. Consequently, the procedures provided by Oak.Schedulers handle these Oak Messages. The Oak.Agent.Tasks.Cycle package provides the kernel's handling support for the cycle related requests.

deadline and execution budget enforcement

To support deadline and execution budget enforcement, each task agent has two cycle timers: one for enforcing deadlines and the other enforcing execution budgets (see *Oak Timers* on page 112 for the description of cycle timers). When enforcing a task's deadline, the deadline timer tracks the absolute deadline of the task, with the timer active while the task performs a cycle. Maintenance of the deadline timer occurs as part of the cyclic task operations provided by the Oak.Agent.Tasks.Cycle package. By contrast, Oak manages budget timers as part of the Oak Run-Loop (see *Managing Timers* on page 117) with the Oak.Agent.Tasks.Cycle.New_Cycle procedure replenishing a task's execution budget each cycle. Oak handles fired cycle timers as part of its interrupt handling facilities as described in *Interrupt Handling* on page 145.

begin cyclic stage
Oak.Agent.Tasks.Cycle

The Begin_Cyclic_Stage procedure handles the Setup_Cycles Oak Message. Simple in nature, the procedure initialises the internal Next_Cycle and Event_Raised components of the Task_Agent: components used to track when a cyclic task's next cycle event can occur and cycle event notification. Of note, the Next_Cycle component sets up the first cycle event for a periodic or yielding task, and defines the earliest time an event may be raised for a sporadic or aperiodic task. The procedure also ensures the deactivation of the deadline timer.

new cycle
Oak.Agent.Tasks.Cycle

The New_Cycle procedure (Figure 32) handles the New_Cycle message on behalf of Oak and administers the bulk of the task cycle behaviour in Acton.

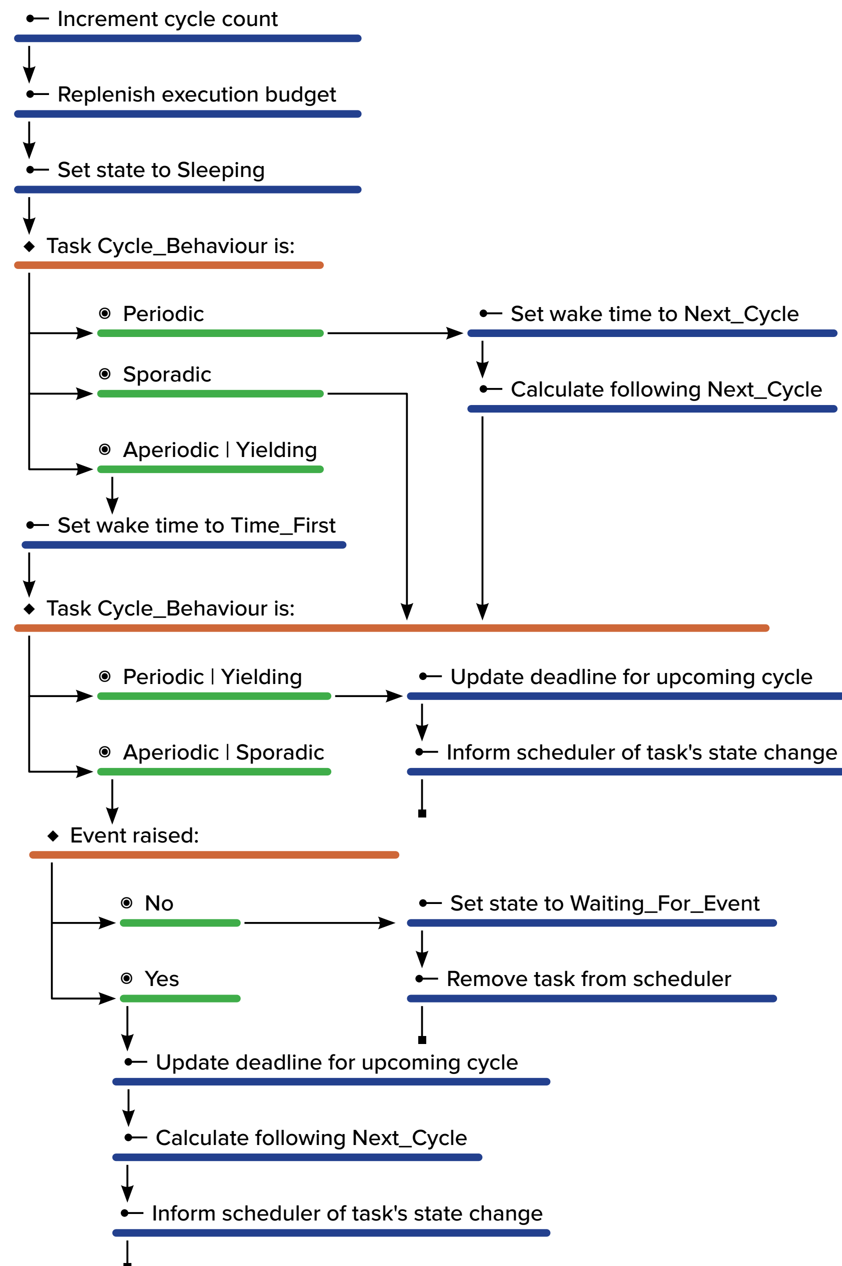


Figure 32
The Oak.Agent.Tasks.Cycle. New_Cycle procedure.

To note:

- All tasks have their state initially updated to Sleeping. Irrespective of a task's cycle behaviour, a Sleeping state will cause its removal from the head of its runnable queue. Whether the task ends up at the tail of its runnable queue or on a sleep queue depends on its wake time.

- The wake time for a periodic task comes from its Next_Cycle component. This wake time is considered the task's run-time generated cycle event.

- For a periodic or sporadic task, the Next_Cycle component contains the time the subsequent cycle may commence: derived from the task's Cycle_Period or Cycle_MIT (Oak collocates both attributes in the Cycle_Period component). A cycle event occurring before this time is considered an early event for sporadic tasks, while periodic tasks skip the next cycle if it is still running the previous cycle.

- Aperiodic and yielding tasks do not use Next_Cycle and instead set their wake time to Time_First since their events do not depend on time. The proceure uses Time_First rather than the current time because while they have the same effect of placing the task on the tail of its runnable queue, Time_First does not come with the overhead of Clock.

- For aperiodic and sporadic tasks, the procedure removes them from their schedulers while they wait for a cycle event. Removal of these tasks ensures schedulers only handle runnable or sleeping agents.

- The absolute deadline of periodic and yielding tasks updates here since a yielding task starts its new cycle immediately and a periodic task's deadline is calculable from Next_Cycle. Aperiodic and sporadic tasks disable the deadline timer until the arrival of the next cycle event since the arrival time of the cycle event is unknown.

- When a task's deadline updates, so does the associated deadline timer.

raise cycle event
Oak.Agent.Tasks.Cycle

The Raise_Cycle_Event procedure handles the Raise_Cycle_Event Oak Message on behalf of Oak, enabling the kernel to allow an aperiodic or sporadic task to commence its new cycle. Figure 33 on page 134 describes its operation.

The procedure begins by enforcing the correct use of the **new cycle** statement, setting the requesting task's state to New_Cycle_Error if the target task possesses yielding, periodic or sequential behaviour. Additionally, the procedure checks to see if the requesting task is triggering an early cycle event. If so, the procedure checks the target task's Early_Event_Response component and responds accordingly.

If the request does not trigger an early cycle event, the procedure adds the target task back to the responsible scheduler agent, updates the task's absolute deadline and, for a sporadic task, calculates the its Next_Cycle time from the task's Cycle_MIT (stored in the task's Cycle_Period component).

When a target task accepts a delayed cycle event, the procedure will mark the task's Event_Raised component if the task has not completed its current cycle. Otherwise, the task is a sporadic task waiting for the expiry of its Cycle_MIT. In this case, the procedure sets the Wake_Time to the expiry of the Cycle_MIT (stored in Next_Cycle) and then treats the task like it has received a non-early cycle event.

update task properties
Oak.Messages
Oak.Agent.Tasks

The Update_Task_Properties message updates the specified task property to the desired value, conveyed through the variant record in Listing 54 on page 134. The Update_Task_Property procedure within the Oak.Agent.Tasks package handles the message, updating the requested property.

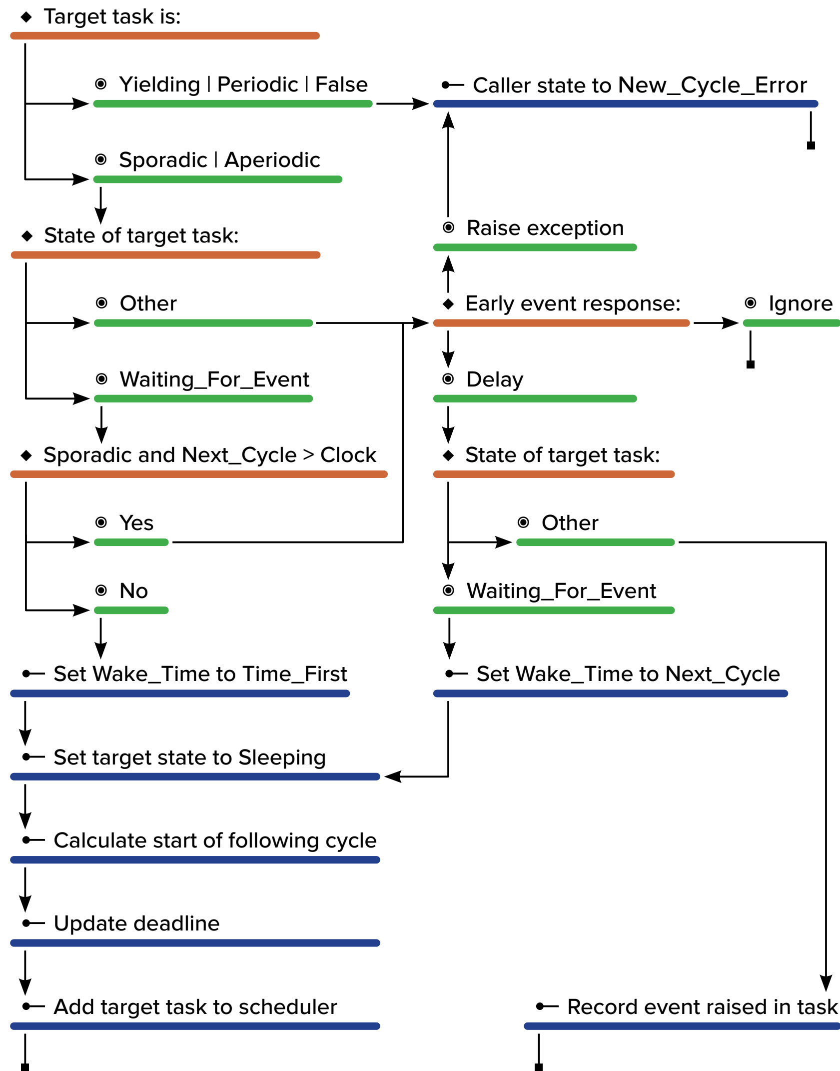


Figure 33
The Oak.Agent.Tasks.Cycle.
Raise_Cycle_Event procedure.

Listing 54
Partial implementation of
Task_Attribute
Oak.Messages
oak-messages.ads

```
type Task_Attribute
  (Property : Task_Attribute_Kind := Cycle_Period) is
record
   case Property is
      when Cycle_Period =>
         Cycle_Period  : Oak_Time.Time_Span;
      when Priority =>
         New_Priority : Any_Priority;
      when Relative_Deadline =>
         Deadline_Span : Oak_Time.Time_Span;
     when Absolute_Deadline =>
         Deadline : Oak_Time.Time_Span;
      when Execution_Budget =>
         Budget : Oak_Time.Time_Span;
      when Deadline_Response | Budget_Response =>
         Response : Ada.Cyclic_Tasks.Event_Response;
      ...
      ...
   end case;
end record;
```

GNAT for Acton Support

Supporting Acton's implementation of the *Cyclic Task Specification*, GNAT for Acton incorporates the new cyclic task syntax and semantics. The compiler's parser and semantic analyser is extended to support the new **cycle** keyword, new_cycle_statement and the extension to the task_body. The compiler also incorporates the *Specification's* new attributes, aspects and pragmas.

Most of the interesting changes to GNAT's expander lies with the task body extension and the generation of a task object's task create call. The simplest addition to the expander is the expansion of new_cycle_statement into the corresponding run-time call:

```
new cycle T; ⇒ Oakland.Tasks.Raise_Cycle_Event (T);
```

More involved is the expansion of the updated task body. Without the optional cyclic section, the compiler uses the existing task body expansion to expand the task body into the corresponding task body procedure by collating the task's sequential_sequence_of_statements and exception handlers to form the generated procedure's handled_sequence_of_statements. If the task body does includes the cyclic section, GNAT for Acton's expander first transforms the task body's two sequence_of_statements and exceptions handlers into a handled_sequence_of_statements containing the cyclic semantics:

```
task body defining_identifier [aspect_specification] is
  declarative_part
begin
  handled_equence_of_statements
end [task_identifier];

handled_equence_of_statements ::=
   sequential_sequence_of_statements
   Oakland.Tasks.Begin_Cyclic_Stage;
   loop
      Oakland.Tasks.New_Cycle;
      cyclic_sequence_of_statement
   end loop;
  [exception
    exception_handler
   {exception_handler}]
```

Listing 55
A cyclic task body after expansion by GNAT for Acton.

Once generated, the expander uses the resulting handled_sequence_of_statements for the task body procedure.

Finally, GNAT for Acton modifies the EXP_CH9.Make_Task_Create_Call procedure to pass the cyclic task attributes defined on a cyclic task's type declaration to the Oak.Agents.Tasks.New_Task procedure.[18]

18. GNAT calls the Make_Task_Create_Call procedure to generate the run-time call to Oak.Agents.Tasks.New_Task as part of the expansion of a task object's initialisation.

Protected Objects

In Acton, protected objects have a kernel representation through protected brokers, providing Oak a means of controlling access at the kernel-level. This implementation of protected objects is unique compared to the other Ada 2012 tasking run-time, GNAT, which builds protected objects on top of the locking facilities provided by an underlying kernel or operating system (see *Protected Objects* on page 87).

Building protected object awareness inside Oak enables Acton to efficiently employ the self-service model for the servicing of queued entry calls. Oak makes the self-service model feasible by allowing access to the protected object to transfer from one task to another: an approach not possible when a protected object uses POSIX or Windows locks to control access. Despite the different approach, Acton loosely follows GNAT's existing techniques to expand protected objects.

Acton Representation

A protected object under Acton consists of two objects: a record object holding the protected object's private components and a protected broker object containing the run-time state of the protected object. Consequently, GNAT for Acton takes the protected type defined in Listing 57 and generates a corresponding value record type containing the protected object's private components:

Listing 56 The value record containing Listing 57's protected object private components.

```
type Protected_NumberV is record
   _protected_broker : Oak.Brokers.Protected_Id;
   The_Number        : Integer;
   SV_Occurred       : Boolean := False;
end record;
```

In addition to the protected object's private components, the generated value record includes a reference to the protected object's broker (whose type, Protected_Broker_Record, is reprinted in Listing 58 from *Listing 49 on page 110*). Among the run-time state recorded, the protected broker contains the address of the protected object's corresponding value record object. This link allows an agent to possess either the protected object's value record or broker ID and still have access to both objects.

A protected broker has three task lists capturing the different stages tasks have when interacting with protected objects: trying to gain access, active within and queued on closed entries. The kernel removes tasks placed on the Contending_Tasks or Entry_Queues lists from their schedulers because they are no longer runnable.

holding a protected object

A task added to Active_Task has exclusive access to the protected object. When a task acquires a protected object, the kernel agent places the object's protected broker on its Active_Protected_Brokers list. The kernel will later dispatch the task when the priority of the broker is higher than other brokers and dispatchable agents. Brokers also have precedence over other agents of the same priority to enforce the Priority Ceiling Protocol.

```
protected type Protected_Number is
   procedure Set_Number (N : Integer);
   function Get_Integer return Integer;
   entry Special_Release;

private
   The_Number  : Integer;
   SV_Occurred : Boolean := False;
end Protected_Number;

protected body Protected_Number is
   procedure Set_Number (N : Integer) is
   begin
      The_Number := V;
   end Set_Number;

   function Get_Number return Integer is
   begin
      return The_Number;
   end Get_Number;

   entry Special_Release
     when The_Number = 5 and Special_Release'Count = 2 is
   begin
      SV_Occurred := True;
   end Release_On_Special_Number;
end Protected_Number;
```

Listing 57
An example of a protected object with entries.

```
type Protected_Broker_Record is record
   Name                   : Agent_Name;
   Name_Length            : Agent_Name_Length;

   Ceiling_Priority       : System.Any_Priority;
   Next_Object            : Protected_Id_With_No;
   State                  : Agent_State;
   Broker_Lock            : Boolean;

   Object_Record          : System.Address;
   Active_Task            : Oak.Agent.Agent_List;
   Contending_Tasks       : Oak.Agent.Agent_List;
   Entry_Queues           : Oak.Agent.Task_Id_With_No;
   Entry_Barriers         : System.Address;
end record;
```

Listing 58
Oak.Brokers.Protected_Objects
oak-brokers-protected_objects.ads

Unlike tasks on other protected broker queues, the kernel agent does not remove tasks that are active inside a protected object from their scheduler agent. Oak does this because the Priority Ceiling Protocol and Oak's task dispatching approach means the task will always dispatch via the protected broker. The technique removes the run-time overhead of changing the priority of the task or removing the task from its scheduler agent.

expansion of protected subprograms

GNAT for Acton uses GNAT's approach for protected subprogram expansion, expanding the public protected subprograms into protected (suffixed by P) and non-protected versions (suffixed by N):

Listing 59 The expansion of Listing 57's protected subprogram definitions.

```
procedure Set_NumberP (_object : in out Protected_NumberV;
                       N       : Integer);
procedure Set_NumberN (_object : in out Protected_NumberV;
                       N       : Integer);

function Get_NumberP (_object : in Protected_NumberV)
  return Integer;
function Get_NumberN (_object : in Protected_NumberV)
  return Integer;
```

Each subprogram takes the protected object's value record, with the procedures having read-write access to the record while functions only have read access; in accordance with ARM 9.5.1. Tasks calling the protected subprograms from outside the protected object call the protected version. These subprograms obtain access to the object, call the corresponding non-protected subprogram and then release access once the non-protected subprogram completes:

Listing 60 Expansion of Listing 57's protected subprogram bodies into their generated protected versions.

```
procedure Set_NumberP (_object : in out Protected_NumberV;
                       N       : Integer) is
begin
   Oakland.Protected_Objects.Enter_Protected_Object
     (PO => _object._protected_agent, Entry_Id => 0);
   Set_NumberN (_object, N);
   Oakland.Protected_Objects.Exit_Protected_Object
     (PO => _object._protected_agent);
end Set_NumberP;

function Get_NumberP (_object : in  Protected_NumberV)
  return Integer is
begin
   Oakland.Protected_Objects.Enter_Protected_Object
     (PO => _object._protected_agent, Entry_Id => 0);
   declare
      R10b : constant integer := Get_NumberN (_object);
   begin
      Oakland.Protected_Objects.Exit_Protected_Object
        (PO => _object._protected_agent);
      return R10b;
   end;
end Get_NumberP;
```

For the generated protected function, GNAT generates a temporary variable to store the return result, enabling the function to exit the protected object before returning.

The generated non-protected subprograms, on the other hand, contain the contents of the protected subprograms: with GNAT copying the body of the protected subprogram over to the generated subprogram as demonstrated by Listing 61. To access the protected object's components inside the non-protected subprograms, GNAT renames the protected object's value record com-

```
procedure Set_NumberN (_object : in out Protected_NumberV;
                       N       : Integer)
is
   R1b         : Protected_ID renames _object._protected_agent;
   The_Number  : Integer renames _object.The_Number;
   SV_Occurred : Boolean renames _object.SV_Occurred;
begin
   The_Number := N;
end Set_NumberN;

function Get_NumberN (_object : in  Protected_NumberV)
  return Integer
is
   R2b         : Protected_ID renames _object._protected_agent;
   The_Number  : Integer renames _object.The_Number;
   SV_Occurred : Boolean renames _object.SV_Occurred;
begin
   return The_Number;
end Get_NumberN;
```

Listing 61
Expansion of Listing 57's protected subprogram bodies into unprotected subprograms.

ponents to local variables of the same name. GNAT also creates a local variable holding the protected object's Protected_ID for when run-time calls require the ID. In addition to the protected subprogram versions, protected subprogram calls internal to the protected object also call the non-protected subprograms.

expansion of protected entries

While Acton expands protected subprograms similar to GNAT, the expansion of protected entries takes a different approach. GNAT's technique results from the need to service a protected entry by a proxy task (as described by Miranda & Schonberg (2004)). Since Oak understands protected entries and can pass ownership of a protected object from one task to another, GNAT for Acton expands protected entries using the same approach as protected subprograms. Thus using the Special_Release protected entry of Listing 57 for example:

```
procedure Special_ReleaseP
            (_object   : in out Protected_NumberV;
             _entry_id : in Protected_Entry_Index) is
begin
   Oakland.Protected_Objects.Enter_Protected_Object
     (PO => _object._protected_agent, Entry_Id => _entry_id);
   Special_ReleaseN (_object, _entry_id);
   Oakland.Protected_Objects.Exit_Protected_Object
     (PO => _object._protected_agent);
end Special_ReleaseP;

procedure Special_ReleaseN
            (_object   : in out Protected_ValueV;
             _entry_id : in Protected_Entry_Index) is
   R6b         : Protected_ID renames _object._protected_agent;
   The_Number  : Integer renames _object.The_Number;
   SV_Occurred : Boolean renames _object.SV_Occurred;
begin
   SV_Occurred := True;
end Special_ReleaseN;
```

Listing 62
The expansion of Listing 57's Special_Release entry.

Differing from the expansion of protected subprograms, the generated procedures include a new _entry_id parameter. Like GNAT, Acton assigns each entry an entry ID to identify it within the run-time, including each member of an entry family. Furthermore, GNAT for Acton generates barrier functions for each entry and entry family:

Listing 63
The expansion of Listing 57's entry barrier.

```
function Special_Release_B1s
           (_object : Protected_ValueV;
            E       : Protected_Entry_Index) return Boolean
is
   R4b         : Protected_ID renames _object._protected_agent;
   The_Number  : Integer renames _object.The_Number;
   SV_Occurred : Boolean renames _object.SV_Occurred;
begin
   return The_Number = 10
          and Oakland.Protected_Objects.Entry_Count (R4b, 0) = 5;
end Special_Release_B1s;
```

The entry Count attribute used in the barrier transforms to a call to Oakland.Protected_Objects.Entry_Count with the ID of the queue which prefixes the attribute.

entry queues

When a task executes an entry call on a closed entry, Oak places the task on the protected broker's Entry_Queues list. The list has a two-dimensional structure (Figure 34), with each task in the primary linked list the head of a separate entry queue. Secondary links from a head task using the Next_Agent component form the queue for that entry. For the Ravenscar Profile, the queued entry call list will only ever have at most one task, allowing Acton to efficiently support both restricted and unrestricted protected objects.

Supporting the entry call queues, each broker has the address of a compiler generated routine enabling the evaluation of a specific protected object's barrier:

Listing 64
Protected object barrier evaluation function for Listing 57.

```
function Special_ReleaseS
           (O : System.Address;
            E : Protected_Entry_Index) return Boolean
is
   type protected_numberVP is access Protected_NumberV;
   _object : protected_numberVP := Address_To_Access (O);
begin
   if E <= 1 then
      return Release_On_Special_Number_B1s (_object.all, 1);
   else
      return Special_Queue_Overflowed_B2s (_object.all, 2);
   end if;
end Special_ReleaseS;
```

The generated function calls the individual barrier evaluation function for the corresponding entry ID. Acton uses a function to map entry IDs to barrier evaluation functions instead of an array because entry families map a series of entry IDs to a single function — which may result in a large array with multiple elements pointing to the same function.

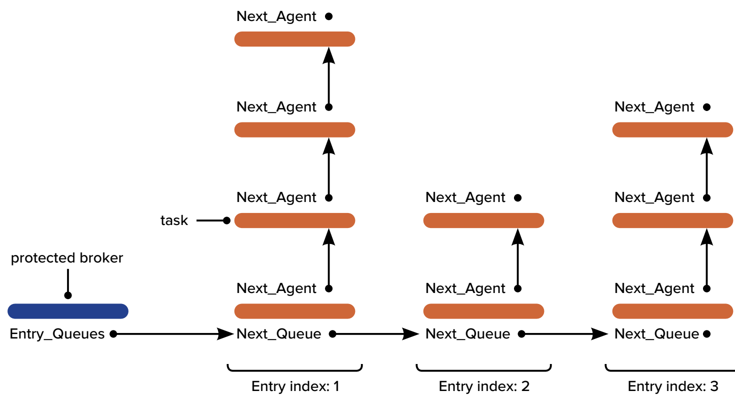


Figure 34 Oak's implementation of protected object entry queues using linked list.

Acton currently has Oak call the protected barrier function to prevent malicious tasks from falsely reporting a closed entry barrier as open. Consequently, a protected object's barrier evaluation function takes not only the entry ID of the barrier but also the address of the protected object's value record. The function uses the record's address because it is the only way to reference a protected object's value record in a common way within the kernel. An address to access conversion facilitates access to the contents of the record.

multiprocessor support

While Acton does not currently offer multiprocessor capabilities, Oak incorporates some early support with respect to protected objects. Oak provides mutual exclusion to a protected broker through the Broker_Lock component; ensuring different processors access the broker in a controlled manner. Tasks are held on the Contending_Tasks list while trying to access the protected object while another task holds the object.

Protected Object Operations

The main protected object operations provided by Acton cover the protected object entry and exit operations. The Oakland.Protected_Objects package provides the run-time interface the compiler uses to interact with Oak. Like many of Oakland's packages, the procedures of Oakland.Protected_Objects primarily consist of making kernel requests and handling the associated responses. The Oak.Protected_Objects package provides the subprograms Oak uses to handle these protected object requests.

protected object entry

A task request entry to a protected object by calling the Oakland procedure Oakland.Protected_Objects.Enter_Protected_Object:

```
procedure Enter_Protected_Object
  (PO             : in Protected_Id;
   Entry_Id       : in Entry_Index := No_Entry);
```

Listing 65 Oakland.Protected_Objects *oakland.protected_objects.ads*

The procedure takes the IDs of the protected object and entry the task wants access to with protected procedure and function identified with an Entry_ID of No_Entry. Calling the procedure causes Oak to receive an Entering_PO Oak Message with the parameters of the procedure. The procedure returns once the task has gained access to the protected object. If the entry request fails, with the task in the Enter_PO_Refused state, the procedure raises Program_Error.

Within Oak, the procedure Oak.Protected_Objects.Process_Enter_Request handles the entry request. While the kernel processes the request, Oak will not dispatch the task until the request completes with the task gaining access to the protected object or the request fails. If unsuccessful, the kernel places the task into the Enter_PO_Refused state; otherwise, the task becomes dispatchable with an Inside_PO state. Figure 35 describes the operation of Process_Enter_Request. To note:

Validate request Oak validates the parameters associated with the entry request, ensuring the requested protected object's broker exists and the entry ID is valid. The check also enforces the Priority Ceiling Protocol, ensuring the priority of task issuing the request has a priority not above the protected object's ceiling priority.

Record request entry ID Enables tasks on a broker's Contending_Tasks list to resubmit their entry request when they are latter admitted to the protected object.

Scheduling Oak conceptually removes tasks from their schedulers when they are inside a protected object (on the broker's Task_Within list) because Oak dispatches these tasks from their protected broker rather than their scheduler agent. However, as an optimisation, Oak does not perform the removal since the kernel implements the Priority Ceiling Protocol: always selecting the task from the broker before the task's runnable queue because of the broker's precedence. Oak, though, does remove tasks on the Contending_Tasks and Entry_Queues lists because these tasks are not dispatchable.

Entry servicing When Oak places a task on an entry queue it causes a re-evaluation of the protected object's entry barriers to check if any barriers have opened (since a barrier may use the queue's Count attribute, as per ARM 9.5.3). An error raised during an entry check causes the entry request to fail.

Exception raised within barriers In this scenario, Oak sets the state of the requesting task and the tasks already on the protected object's entry queues to Entering_PO_Refused. The procedure then purges the entry queues as per ARM 9.5.3:7 and places the formally queued tasks back on their scheduler agents.

Broker lock On multiprocessor systems, Oak requires kernel agents accessing a protected broker object to hold its Broker_Lock.

protected object entry

A task requests its exit from a protected object by calling the Oakland procedure Oakland.Protected_Objects.Exit_Protected_Object:

```
procedure Exit_Protected_Object (PO : in Protected_Id);
```

The procedure Oak.Protected_Objects.Process_Exit_Request, shown in Figure 36 on page 144, handles the exit request within Oak. It verifies that the exiting task is inside the protected object before transferring the protected broker to a task on an entry queue with an open barrier, if such a task exists. Otherwise, Oak deactivates the broker by removing it from the kernel agent's Active_Protected_Brokers list.

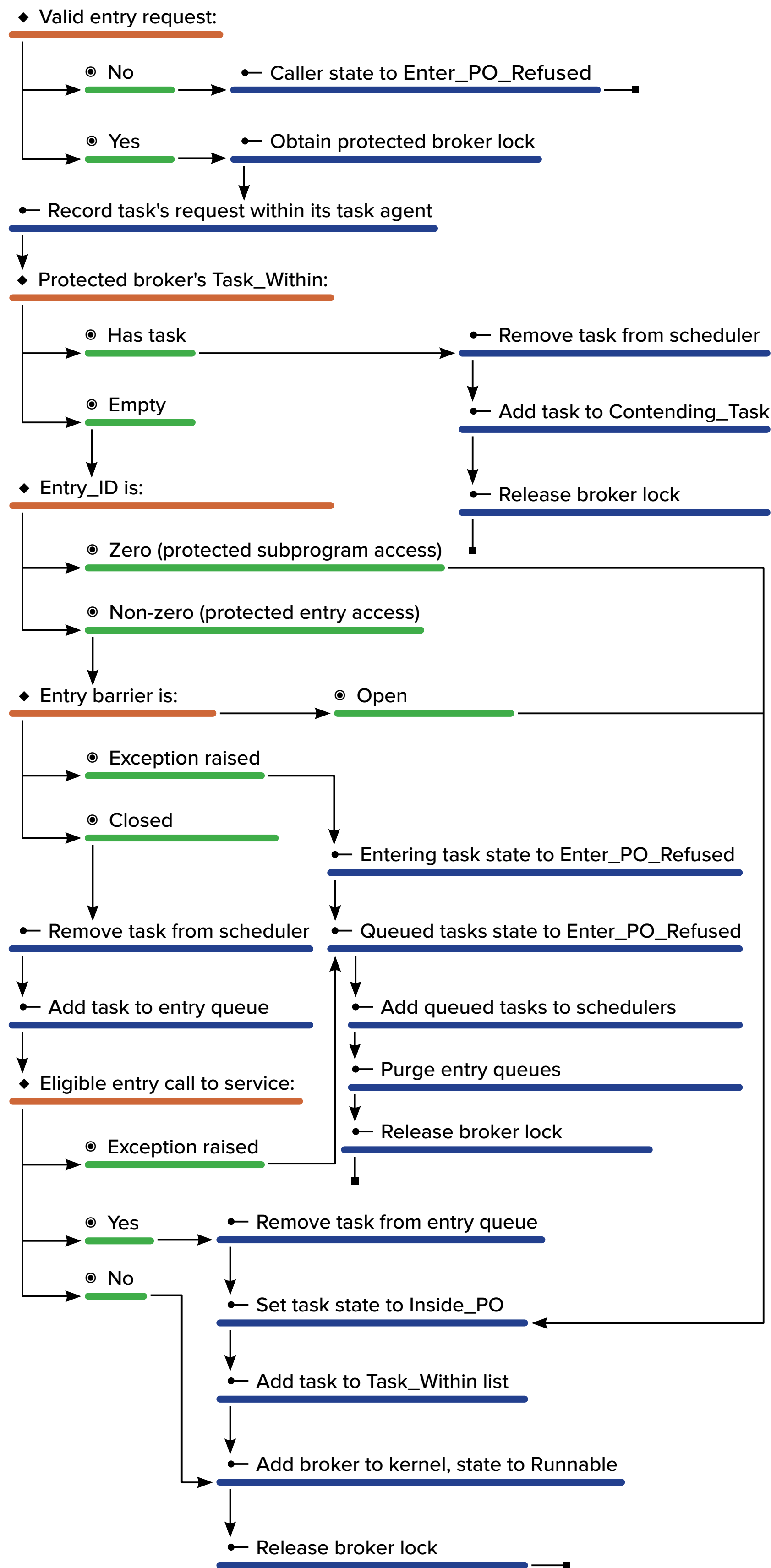


Figure 35
The Oak.Protected_Objects.
Process_Enter_Request
procedure.

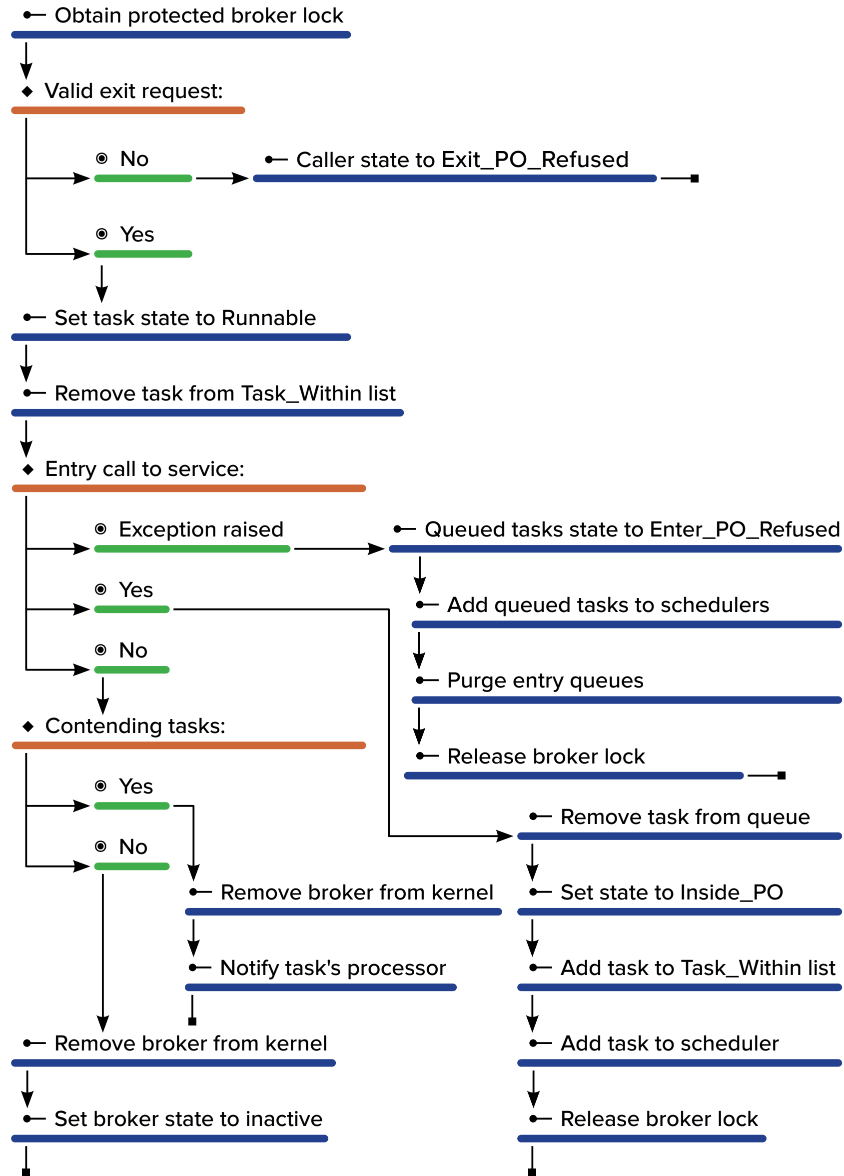


Figure 36
The Oak.Protected_Objects.
Process_Exit_Request
procedure.

On multiprocessor systems, tasks from other processors will have queue on the broker's Contending_Tasks list while the exiting task had operated inside the protected object. Once Oak incorporates support for multiprocessor systems, it is anticipated the Process_Exit_Request will use a future inter-kernel messaging system to pass the protected broker to the kernel responsible for the task selected from the broker's Contending_Tasks list.[19] Once the target kernel agent receives the broker, the kernel will extract its task from the broker's Contending_Tasks list and add the task back to its scheduler. Subsequently, the target kernel agent will re-issue the task's protected object entry request by calling Oak.Protected_Objects.Process_Enter_Request, using the entry request parameters stored within the task's agent. During the handover of the protected broker between kernel agents, the broker remains locked with ownership of the lock being handed over as part of the exchange.

19. Assuming a multiprocessor Oak will have a separate kernel agent running on each processor and an inter-kernel messaging system facilitating the exchange of messages between kernels.

Interrupt Handling

A user can attach user-defined handlers to two sources of interrupts in Acton: external interrupts and Oak Timers. External interrupts originate from outside the processor core via an interrupt controller. By contrast, Oak Timers provide an abstraction over the system timer: sharing the same interrupt source but trigged when their specified time has passed. Three forms of Oak Timers exist: Cycle_Timers, Timing_Event_Timers and Scheduler_Timers. To adapt to the needs of the different timer kinds, the single Oak Timer type uses a variant record to provide specific support for each timer kind (as described in *Oak Timers* on page 112). The system timer takes a form dictated by the facilities provided by the microcontroller; the timer may exist within software, via external hardware or within the processor core itself.

Aside from the Oak Timers abstraction, Acton does not expose a processor core's internal interrupt sources to the user, reserving them for internal use. Internal interrupts link closely to the functionality of Oak and are processor specific— details not explored here. By contrast, Oak adopts a common system handler for external and system timer interrupts: the kernel runs to dispatch the appropriate interrupt agent with the user's interrupt handler.

User Handlers

Ada's protected procedure handlers and timing events provide the user-defined interrupt handlers for external interrupts and Oak timers respectively. Oak also uses Oak Timers to implement execution budgets and deadlines for cyclic tasks, with the handling of these timers determined by the Event_Response type associated with each timer (as defined by the *Cyclic Task Specification*). Finally, the handling of scheduler timers differs from the other timers as they support Oak's operation and not the user's (see *Management of Scheduler Agents by Oak* on page 120).

cycle timers

Cycle timers support cyclic tasks. Each task agent contains two cycle timers supporting the *Cyclic Task Specification's* responses to missed deadlines and budget exhaustions (*Cyclic Tasks* on page 130). The user specifies the handling behaviour of these timers through the Deadline_Response and Budget_Response task attributes, with Oak only supporting the No_Response and Handler options.[20] A cycle timer uses the ceiling priority of the handler's protected object when the timer uses the Handler response; otherwise, the timer uses the cyclic task's priority. Management of these timers falls on the Oak.Agent.Tasks.Cycle.New_Cycle procedure and Oak's run loop (*Cyclic Tasks* on page 130 and *The Oak Run-Loop* on page 114).

timing events

Oak uses timing event timers to implement Ada's timing events (ARM D.15), with the user interface provided through the Ada.Real_Time.Timing_Events package. Internally, an Oak Timer captures the timing event semantics, leaving the Timing_Event type to just reference the timer (Listing 66). The Set_Handler procedures send a Set_Timing_Event_Handler request to Oak, with Oak assigning a new Oak Timer to the timing event object if it lacks one. The associated Oak Timer assumes the Interrupt_Priority'Last priority as per ARM D.15.14.

20. Since Oak currently lacks asynchronous transfer of control and task abortion.

Listing 66
Ada.Real_Time.Timing_Events
ada-real_time-timing_events.ads

```
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
   type Timing_Event is tagged limited record
      Timer : Oak.Timers.Oak_Timer_Id;
   end record;
end Ada.Real_Time.Timing_Events;
```

The Cancel_Handler procedure sends Oak a Cancel_Timing_Event request causing Oak to remove the associated oak timer, while the two functions directly query the Oak Timer. Since Acton does not currently support finalisation, Acton only supports timing events declared at the library-level.

external interrupts

Ada's protected procedure handlers (ARM C.3.1) allow users to attach user-defined handlers on external interrupts. Support for these handlers lie within the Oakland, Oak and OHAL packages. Oakland provides the Oakland.Interrupts package (Listing 67) containing the Attach_Handlers procedure. GNAT generates a call to the procedure as part of the expansion of a protected object type declaration with attached handlers. The procedure takes an array of handlers because a protected object may contain multiple exception handlers. The Attach_Handlers procedure sends Oak an Attach_Interrupt_Handler message to connect the handlers to their external interrupts.

The Oak.Interrupts package (Listing 68) handles the Attach_Interrupt_Handlers Oak Message and provides processor-independent interrupt types. Since interrupts are micrcontroller specific, the actual attachment occurs within OHAL's Oak.Processor_Support_Package.Interrupts package (Listing 69) through the Attach_Handler procedure. The package also provides the processor-specific interrupt data structures, with interrupt handlers typically stored in an array mapping interrupt IDs to handlers. While normally machine independent, OHAL houses the array to permit Oak to write the array into Flash memory during system initialisation — useful for microcontrollers with static handler assignment and significantly more Flash memory than RAM. The processor-dependent Attach_Handler procedure also takes care of configuring the interrupt controller for the corresponding interrupt: for example, assigning the priority of the interrupt and enabling the interrupt within the interrupt controller.

```
package Oakland.Interrupts with Preelaborate is
   procedure Attach_Handlers
     (Handlers  : in Interrupt_Handler_Array);
end Oakland.Interrupts;
```

Listing 67
Oakland.Interrupts
oakland-interrupts.ads

```
package Oak.Interrupts with Preelaborate is

   type Interrupt_Handler_Pair is record
      Interrupt : External_Interrupt_Id;
      Handler   : Parameterless_Handler;
   end record;

   type Interrupt_Handler_Array
     is array (Positive range <>) of Interrupt_Handler_Pair;

   procedure Attach_Handler
             (Handler : in Interrupt_Handler_Pair);

end Oak.Interrupts;
```

Listing 68
Oak.Interrupts
oak-interrupts.ads

```
package Oak.Processor_Support_Package.Interrupts
  with Preelaborate is

   subtype External_Interrupt_Id is MPC5554.INTC.INTC_ID_Type;
   Default_Interrupt_Priority : constant Interrupt_Priority :=
                                         Interrupt_Priority'Last;

   type Parameterless_Handler is access protected procedure;

   procedure External_Interrupt_Handler
             (Interrupt_Id : External_Interrupt_Id);
   procedure Initialise_Interrupts;
   procedure Complete_Interrupt_Initialisation;

   procedure Attach_Handler (Interrupt : External_Interrupt_Id;
                             Handler   : Parameterless_Handler;
                             Priority  : Interrupt_Priority);

   function Current_Interrupt_Priority return Any_Priority;
   function Get_External_Interrupt_Id
     return External_Interrupt_Id;

end Oak.Processor_Support_Package.Interrupts;
```

Listing 69
Oak.Processor_Support_Package.
Interrupts
oak-processor_support_pack-age-interrupts.ads
(MPC5554 PSP)

Handling Interrupts

Acton's splits the handling of interrupts between the kernel and its interrupt agents. Oak manages the processor core and the interrupt controller: determining which interrupt was raised, acknowledging and clearing the interrupt source within the core or controller, and reacting to the interrupt. By contrast, interrupt agents run user defined handlers.

In response to the raising of an interrupt, the processor's kernel agent executes a new iteration of the kernel's *Oak Run-Loop*. The *Run-Loop* enters with a run

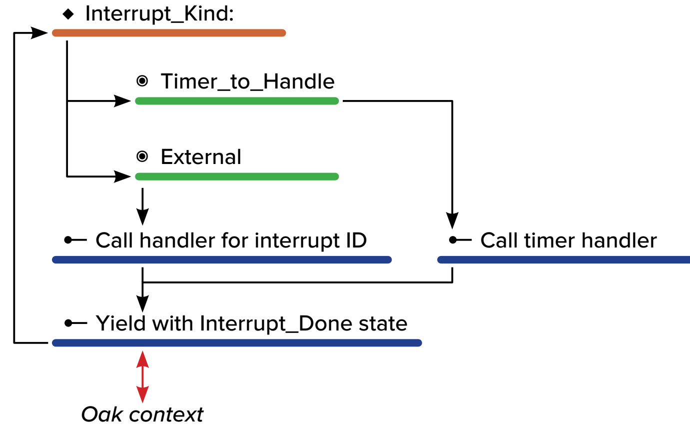


Figure 37
Interrupt agent run-loop flow.

reason of External_Interrupt or Oak_Timer depending whether the interrupt came from the external interrupt controller or the system timer. Figure 38 describes the handling of an interrupt within the *Run Reason* stage of the *Oak Run-Loop* (*Run Reason* on page 115). Note Oak tolerates the system timer firing before the earliest Oak Timer to support microcontrollers whose system timer implementation lacks the range to cover Ada.Real_Time.Time_Span.[21]

Oak handles the firing of scheduler timers within its management of scheduler agents (see *Management of Scheduler Agents by Oak* on page 120). The other sources of interrupts — external interrupts, timing event timers and cyclic timers — have their user provided handlers executed inside interrupt agents. Oak assigns an individual interrupt agent for each interrupt priority, permitting the nesting of interrupts. When an interrupt with a user-define handler occurs, the kernel dispatches the interrupt agent associated with the priority of the interrupt. While handling, the interrupt agent takes precedence over any other agent at that priority level.

To support the interrupt agent, its kernel object (of type Interrupt_Agent) contains three components set by Oak during an interrupt and used by the agent to fetch the associated handler (*Interrupt Agents* on page 108). As shown in Figure 37, the run-loop of the interrupt agent calls the associated handler and subsequently returns control back to the kernel through the Interrupt_Done message. On receiving the message, the kernel agent deactivates the interrupt handler, completing the interrupt.

Rationale

Four reasons motivate Acton's use of interrupt agents: they provide a separate stack for each interrupt priority; allow interrupt handlers to call kernel services; reduce machine dependent code within OHAL; and enable the kernel to consist of a common run-loop.

Providing a separate stack for each interrupt priority removes the need for each task to reserve a portion of its stack for interrupt handlers. The effect of the latter approach on task stack size becomes particularly pronounced when supporting nested interrupts: every task's stack has to support the worst-case interruption scenario, a scenario which can only affect one task at any one time.

21. For example, the MPC5554 32-bit decrementer is smaller than the 64-bit Time_Span.

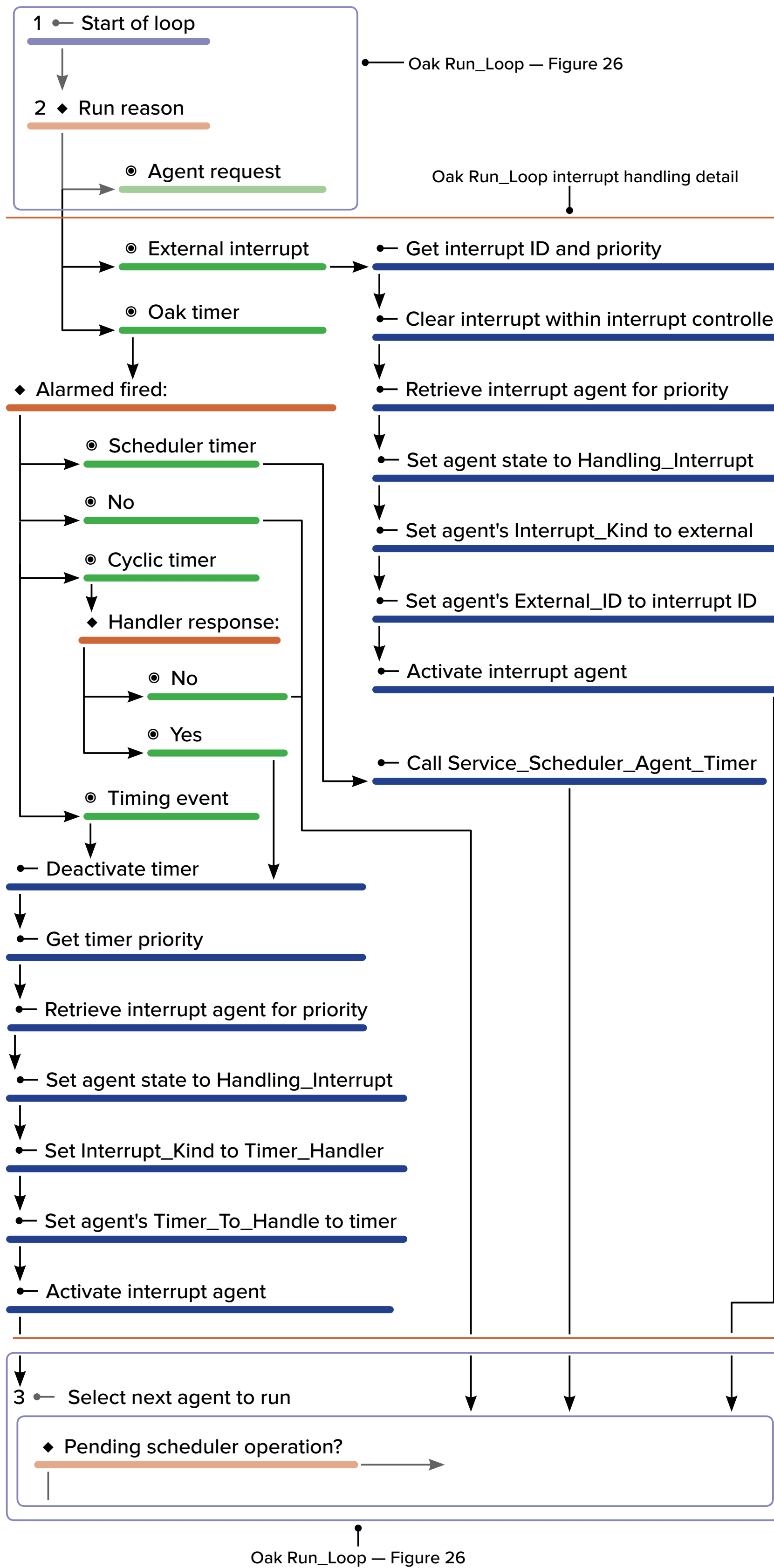


Figure 38
Handling of interrupts within Oak Run-Loop.

Executing interrupt handlers within their own agent allows handlers to make kernel calls: for instance, change task parameters or releasing blocked tasks. Acton's design of making kernel calls through Oak Messages, and the manipulation of a task's agent objects in response, assumes a one-to-one mapping of a context to an agent. Thus, allowing an interrupt handler to execute within the context of a running agent would corrupt the task's kernel object when making a kernel call.

Handling interrupts through a kernel's *Oak Run-Loop* helps reduce machine dependent code within OHAL by permitting the sharing of a common context switching routine between the different interrupt sources invoking Oak: software call, timer and external interrupts. Reducing the size of OHAL simplifies testing and porting Acton to a new microcontroller. Running interrupts through the kernel's *Run-Loop* also simplifies the design and maintenance of the kernel since the kernel has only one entry and exit point. Furthermore, it enables Oak to use its existing timing facilities to allow it to correctly apportion the execution time of interrupts with no extra code and to ensure a low priority timer does not interrupt an active interrupt.

Conclusion

Acton possesses a unique design based on the construction of a real-time executive supporting not only Ada 2012's real-time tasking facilities, but also providing scope and flexibility to experiment and extend them. Freed from the constraints of an existing run-time, Acton provides a greater subset of Ada's tasking facilities than the existing Ada executive, GNAT for Bare Boards, while still targeting microcontrollers like Freescale's MPC5554, Atmel's SAM7S and STMicroelectronic's STM32F4. Extended support includes protected objects with multiple unbounded entry queues and different task dispatching policies. The flexibility provided by the architecture of Acton facilitates the incorporation of the *Cyclic Task Specification*, including seamless support for execution servers within its scheduling model.

Starting from a clean sheet with control over both the kernel and the run-time allows Acton to focus on the needs of Ada and real-time users without the constraints imposed by pre-existing run-times or kernels. Consequently, Acton implements the bulk of Ada's dynamic semantics in its kernel rather than a run-time library. In particular, its approach facilitates kernel-level support for protected object with a lock-free design; enabling the use of the self-service model for protected entries, simplifying timing analysis and enforcement.

Acton currently supports three different microcontrollers. Development originated on the Power Architecture-based MPC5554 by Freescale, followed by subsequent ports to two ARM-based microcontrollers: Atmel's AT91SAM7S — the heart of LEGO's Mindstorm NXT control brick and based on ARM7 — and STMElectronic's STM32F4 — a microcontroller with GNAT for Bare Boards support and based on the Cortex-M4. Together, the three microcontrollers represent three different microcontroller designs: possessing different processor architectures,[22] timing and timer sources, and interrupt facilities; together demonstrating the portability of Acton.

22. The ARM7 and Cortex-M4, while ARM-based, are several generations apart and have considerable architecture differences.

Furthermore, at one stage of Acton's development the real-time executive was running on an Atmel AVR 8-bit microcontroller. While demonstrating Acton's portability, the lack of a suitable monotonic time source and issues storing time in a 32-bit integer[23] saw the AVR port abandoned. The port though demonstrates the unsuitability of AVR microcontrollers for real-time systems and indicates their strength lies in event-response systems. This observation presents a future opportunity to develop an Event Acton derivative: using Acton's flexibility to strip time from the executive to produce an event-driven executive targeting event-response systems.

23. 32-bit operations are expensive on an 8-bit microcontroller, consume precious memory, and does not provide sufficient range for the Ada.Real_Time.Time type (as per ARM D.8), with Time on the AVR port sporting a range of about 30 days. While storing time in a 64-bit integer would solve the latter, it would compound the former two issues.

6

Chapter 6

Review of Acton

Prior to Acton, the only Ada 2005/2012 tasking run-time available for 32-bit microcontrollers was GNAT for Bare Boards: an Ada executive offering a subset of Ada's tasking run-time conforming with the Ravenscar Profile. Chapter 4 reviewed the real-time limitations of GNAT-BB, detailing how its scheduling, protected object and execution time operations negatively affected the development of reliable and maintainable real-time systems. The findings prompted the development of Acton: an executive satisfying the needs of real-time users while incorporating the *Cyclic Task Specification*. Notably, Acton provides greater flexibility for real-time and embedded users by embracing Ada tasking features outside the Ravenscar Profile, including wider protected object support and support for different task dispatching policies. However, while Acton tries to embrace a greater subset of Ada, the real-time executive lacks GNAT-BB support for multi-processors.

Fundamentally, different design decisions underlie Acton and GNAT-BB: Acton is a new real-time executive designed to meet the needs of real-time users while GNAT-BB sought to reuse an existing Ada run-time. Acton's new design also facilitated the trial of different ideas, for example removing scheduling from the kernel while integrating protected object support. This chapter reviews the design decisions of both Ada executives, focusing on the two areas of concern for users: the memory and processor overheads imposed by the executives; and the complexity they bring to timing and schedulability analysis.

Platforms and Setup

GNAT-BB started on a SPARC-based ERC32 emulator (la Puente, Ruiz & Zamorano, 2000A) before being ported to a range of SPARC, ARM and Power Architecture microcontrollers (AdaCore, 2013B). Acton by contrast began on the Power Architecture MPC5554 and then subsequently ported to a range of ARM-based microcontrollers. The range of different microcontrollers supported by both Ada executives demonstrates their portability: neither possess a design limiting them to a single processor architecture or microcontroller.

To provide a comparison of the memory and processor overheads between the two Ada executives, both have been configured for the Freescale MPC5554 (Power E200), Atmel AT91SAM7S (ARM7) and STMicroelectronics STM32F4 (Cortex M4) microcontrollers. AdaCore has provided the AT91SAM7S support for GNAT-BB, with the microcontroller distinguished by being at the heart of LEGO Mindstorms: enabling the development of a simple, but non-trivial, real-time application (AdaCore, 2013A). The subsequent availability of a STM32F4 port of GNAT-BB motivated Acton's support for the microcontroller, providing a comparison on a modern ARM platform. The MPC5554 port of GNAT-BB was adapted for this thesis from an existing MPC5566 port, extending the executive comparison to a second processor architecture.

Configuration flexibility exists within both Ada executives. To ensure a fair comparison, this chapter aims for a similar configuration across the two executives and uses a consistent set of compiler options to compile them.

compiler configuration

GNAT suppresses all language-defined run-time checks when compiling packages within the Ada, Interfaces and System package hierarchies. Since GNAT-BB implements its kernel inside the System.BB hierarchy, the kernel similarly lacks these checks. GNAT disables the language-defined run-time checks because they add overhead to the run-time code, imposing significant penalties due to the frequency of execution. Consequently, to reduce the overhead for proven code, GNAT disables the implicit run-time checks in favour of explicit checks at the boundary of the run-time to enforce its correct operation (FSF, 2014).

Acton follows the same route with explicit checks on the values passed to the kernel via Oak Messages and suppresses all language-defined run-time checks (using the –gnatp flag when compiling). All test programs used in this chapter apply the default set of language-defined checks defined by GNAT.

Both the executives and the test programs use GNAT's –O2 optimisation level as recommended by the GNAT Pro User guide (AdaCore, 2015D) and discard object names. Finally, Acton takes advantage of link-time optimisations on the ARM microcontrollers to compile the executive.

executive configuration

Acton and GNAT-BB have been setup to share similar configurations across each microcontroller. Common is the use of the FIFO_Within_Priorities task dispatching policy, execution time tracking and the collocation of the primary and secondary stacks for all tasks (except for GNAT-BB's main task which uses a separate 1 KB secondary stack). The default stack size for each task is 1 KB. Table 2 to Table 4 document the microcontroller and executive specific configurations.

AT91SAM7S	Acton	GNAT-BB
Interrupt priorities	1	
Interrupt stack arrangement	dedicated stack	dedicated stack + current task stack
Dedicated interrupt stack size	512 bytes	128 bytes
Interrupt vector table location	RAM	
Main task stack size	4 KB	
Kernel stack size	256 bytes	—

Table 2 Executive configuration for the Atmel AT91SAM7S.

STM32F4	Acton	GNAT-BB
Interrupt priorities	15	
Interrupt stack arrangement	stack per interrupt priority	
Interrupt stack size	1 KB per priority	
Interrupt vector table location	RAM	
Main task stack size	4 KB	
Kernel stack size	312 bytes	–

Table 3 Executive configuration for the STMicroelectronic STM32F4.

MPC5554	Acton	GNAT-BB
Interrupt priorities	15	
Interrupt stack arrangement	stack per interrupt priority	shared interrupt stack
Interrupt stack size	1 kb per priority	8 kb shared
Interrupt vector table location	flash	ram
Main task stack size	4 kb	
Kernel stack size	388 bytes	—

Table 4 Executive configuration for the Freescale MPC5554.

timing measurements

Test programs undertaking timing measurements invalidate the microcontroller caches, if present, at the start of each timed run to provide a worst-case scenario where the cache does not contain the program and kernel. The STM32F4 and MPC5554 only cache Flash memory, with caching of SRAM unavailable on the STM32F4 and disabled on the MPC5554. For each microcontroller, the timing source for the measurements is:

MPC5554 *Time Base Lower Register* (32-bits), running at core frequency.

STM32F4 *Cycle Count Register* (32-bits), running at core frequency.

AT91SAM7S *Timer Counter* peripheral (16-bit), clocked at half the core clock.

The timing measurements are obtained by running test programs that perform the operation under examination 100 times. These programs are run three times, with the largest recorded time reported.

Kernel Design

Differing design philosophies underlie Acton and GNAT-BB, producing contrasting run-time overheads and usability. Their biggest differences lies in how the executives arrange their kernel stack space and store their kernel objects.

Kernel stack space

Both Acton and GNAT-BB implement a single address space. However, Acton's original design was motivated by a desire to support virtual memory to offer protected memory spaces for each task through the memory management unit found in many microcontrollers. While Acton has not ended up including virtual memory support, a consequence of this early design required Acton to place the kernel into its own separate context. Acton required this because protecting tasks also required protecting the data structures of the kernel. Consequently, modification of these data structures occurs within the context of the kernel.

Conversely, the design of GNAT-BB started with a single address space from the outset. Consequently, GNAT-BB takes a different approach and does not use a dedicated kernel thread: instead, kernel operations operate within the context of the current task through user callable subprograms. Kernel protection procedures — System.BB.Protection.Enter_Kernel and Exit_Kernel — bracket all kernel subprograms to enforce kernel consistency. Context switching only occurs between tasks as part of the Exit_Kernel procedure as required.

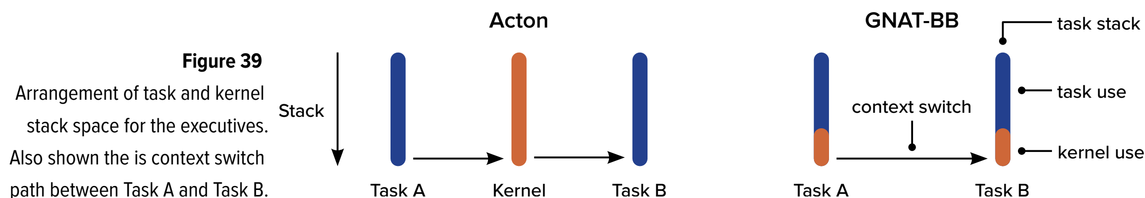


Figure 39
Arrangement of task and kernel stack space for the executives. Also shown the is context switch path between Task A and Task B.

These two kernel approaches make different trade-offs: processor cycles versus stack usage as shown in Figure 39. Acton incurs large processor overheads when carrying out a kernel operation due to the context switches between task and kernel. While GNAT-BB significantly reduces execution overheads by having tasks perform their own kernel operations, the executive requires each task to reserve some of its stack space for the kernel's use. Since on a single processor only one task will have an active kernel operation at any point in time, the design of GNAT-BB means the reserved kernel space in other tasks is effectively wasted space. Acton with its kernel agent does not suffer from this problem as the kernel has its own stack. Additionally, if the kernel demands on the stack increases due to kernel maintenance, Acton only needs to increase the size of the kernel agent stack as opposed to GNAT-BB which requires analysis of each task's stack size to ensure the new memory requirements fit.

Figure 40 and Figure 41 document the kernel overheads for the executives. Figure 40 documents the stack space each task on a GNAT-BB system has to reserve for the kernel and the stack space required for an Acton kernel agent. The total stack space occupied by GNAT-BB's kernel on a given system becomes the product of the number of task, whereas on Acton this space is fixed. Note the effect on GNAT-BB's stack requirements by including the Ada.Execution_Time package — enabling execution time tracking within the kernel. This package

Acton's kernel has a higher stack footprint, but only repeated once for each kernel agent.

Whereas GNAT-BB imposes its kernel stack footprint on each task, with execution time tracking imposing a greater burden than without.

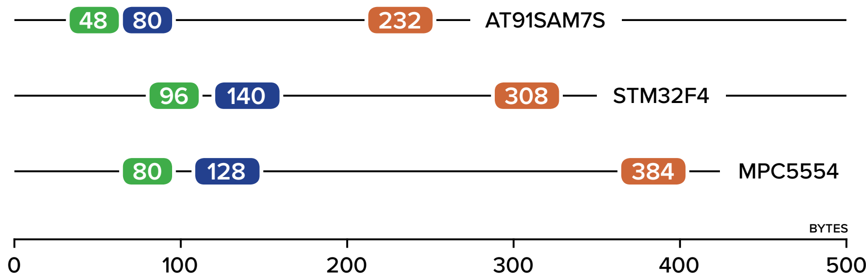


Figure 40 Stack space required by the kernel across three micrcontrollers. For GNAT-BB this value the reserve space per task and for Acton the reserved space per kernel agent.

A dedicated kernel thread increases Acton's kernel operation overhead over GNAT-BB.

For example, the overhead to access a protected procedure.

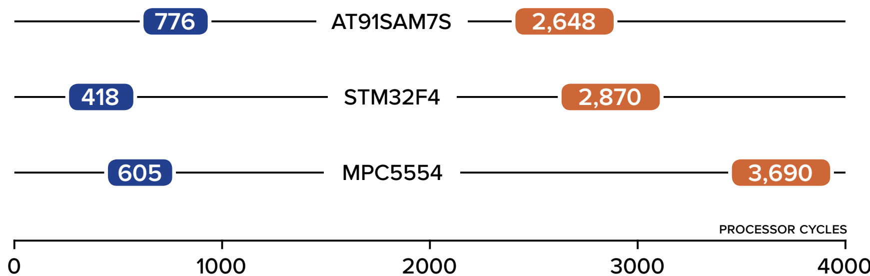


Figure 41 Overhead in processor cycles to access a protected object without contention.

modifies the kernel and demonstrates the effect kernel modification has on the memory demands of the kernel and the impact it has on task stack usage.

Figure 41 by contrast presents the processor cycles taken to perform a trivial, uncontested protected procedure call. Such a call is all run-time overhead and consists of two kernel operations: one to enter the protected object and another to exit. While Acton and GNAT-BB implement protected objects differently, the presented results are indicative of the kernel overhead since the operations themselves are trivial.[24]

Predominately, the gulf between the two executives results from Acton requiring a context switch to the kernel for every kernel operation: swapping the current task's register set for the kernel's. Acton has to swap up to 75 registers each context switch on the MPC5554, while the AT91SAM7S and STM32F4 microcontrollers only have to switch a smaller number of 30 registers, partly explaining the overhead difference between the microcontrollers. For the presented operation, Acton performs four context switches while GNAT-BB none.

The larger overhead incurred by accessing Acton's kernel operations, however, is not all due to the context switching mechanism: agent execution time tracking imposing its own notable overhead. While GNAT-BB can track the execution time of tasks, the executive cleverly only enables execution time tracking when a program references the Ada.Execution_Time package. To provide a

24. The the entry and exit calls for Acton require two pointer operations while GNAT-BB simply records the task's inherited priority.

fair comparison between the two executives, the results presented in Figure 41 include execution time tracking. Despite this, the results from this example do not include any execution time overhead because GNAT-BB only acounts execution time during context switches. Since no context switch occurs in the above example, GNAT-BB is freed from performing expensive execution time calculations. This differs from Acton, which tracks the execution time of the kernel separately from tasks and thus needs to perform two execution time updates each context switch to the kernel.

The expense of execution time tracking comes from the Ada.Real_Time.Clock call (Figure 42) and the overhead of doing 64-bit arithmetic on a 32-bit platform (since the executives represent Ada.Real_Time.Time as 64-bit integers). Its impact on the execution overhead of kernel calls is best demonstrated by the overhead of running a protected procedure handler in response to an interrupt, since under GNAT-BB this invokes a context switch. Using the protected procedure from Figure 41 as the protected handler, Figure 43 documents the interrupt overhead for Acton and GNAT-BB, demonstrating the effect execution time tracking has on the kernel overhead of GNAT-BB.

Combined with the results from Figure 41, Figure 43 shows the effects switching contexts and execution time tracking has on kernel call overheads. For example, the time to complete the protected procedure via an interrupt on GNAT-BB can almost double the normal protected call time due to the context switching required. Furthermore, Figure 43 shows execution time tracking can impose up to 50% overhead on top of context switching. In some respects, Oak compares favourable to ORK since it only takes 1.5 to 2.5 times as long to perform twice as many context switches and execution time updates as ORK.[25]

The downside is Acton has to make these extra context switches whereas GNAT-BB does not and the design of Acton dictates the execution time tracking of the kernel. Separate tracking of Oak's execution time does have a benefit though: enabling the execution time of a task to contain only the actions performed by the task itself; valuable since the length of kernel operations can be effected by the number of tasks operating in the system.

As a final note, Acton can become more competitive to GNAT-BB on advanced microcontrollers like the MPC5554 by utilising its L1 cache inventively. The MPC5554 permits accessing the cache as RAM, enabling the kernel stack and key data structures, like the agents pools, to be stored in the low latency memory. Together with pinning key kernel subprograms in the cache, this setup allows Acton to complete a protected handler in 2,305 cycles.

Kernel Storage

Acton and GNAT-BB adopt different approaches to allocating and referencing their kernel objects. GNAT-BB, which only has tasks, allocates its kernel objects statically when the program compiles as part of the expansion of tasks. References to these objects use memory pointers. Acton, as part of an effort to avoid using pointers within the kernel, allocates agent, broker and timer objects from storage pools. As implemented, a storage pool consists of a static array allocated during the compilation of the kernel, with objects referenced via array indexes. Acton's approach offers four benefits:

25. Also note the similar overheads for Acton calling the protected procedure verses the protected handler: a result of both operations sharing similar handling behaviours.

A call to Ada.Real_Time.Clock is not trivial.
With the upper-bound of the call lower on Acton than GNAT-BB.

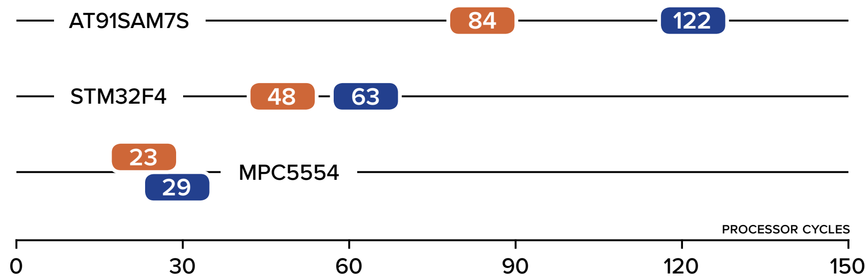


Figure 42 Upper-bound on a call to Ada.Real_Time.Clock in processor cycles

Increased overheads occur when invoking a protected handler on GNAT-BB with execution times tracked compared to when they are not.
While Acton's overhead is similar to a protected procedure call.

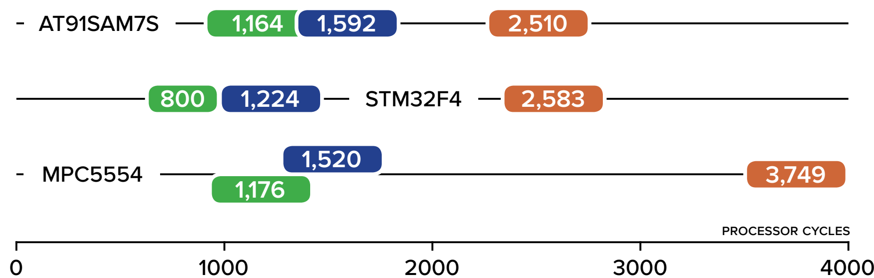
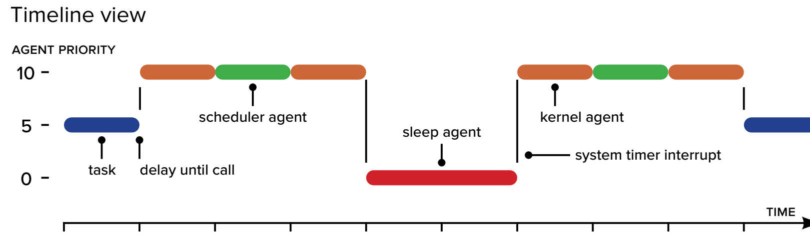


Figure 43 Overhead of a protected procedure handler via an interrupt.

- Removes the null pointer and its handling (such as guards preventing null pointer dereferencing, as all IDs map to an object within the pool).
- Using the No_ identifier to represent non-selection from the pool (such as No_Agent) enables Acton to assign additional meaning to the No_ identifier as appropriate. For example, Acton assigns the sleep agent to the No_Agent identifier, since the selection of No_Agent to dispatch entails the dispatching of the sleep agent. This ability to assign additional meanings simplifies Acton's design and maintenance.
- Allows object references to use smaller hardware types. For example, pointers are 32-bits on the microcontrollers used in this chapter. If using less than 255 agents on Acton, an 8-bit unsigned type provides sufficient storage to represent agent IDs. Reducing the size of object references brings down memory requirements, and can increase performance by reducing cache demand and allowing the fetching of adjacent object references in a single read operation.
- Storage pools provide good data locality and makes it simple to retrieve all the kernel object for reference during debugging.

Working against these benefits, Acton allocates the maximum number of kernel objects at its compile time. Consequently, an Acton-based application not using all of the allocated kernel objects wastes memory. While rebuilding Acton is possible to allocate the right number kernel objects — by updating the Oak.Project_Support_Package package — this approach differs from the simplicity of GNAT-BB: only allocating the resources it needs at program compile-time.

Scheduling and Task Dispatching



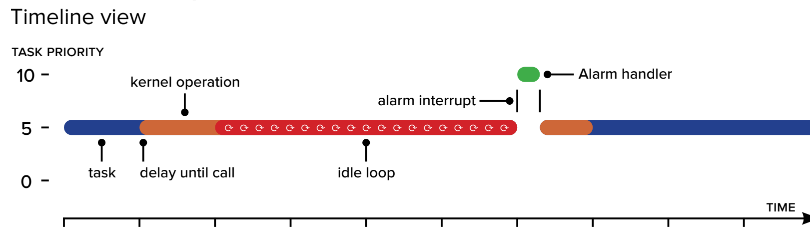


Figure 44
Timeline for a task calling a delay until statement and then waking. Note for GNAT-BB: the kernel operation occurs in the task's context with interrupts disabled and the alarm handler may only require a partial save of the interrupted task's context.

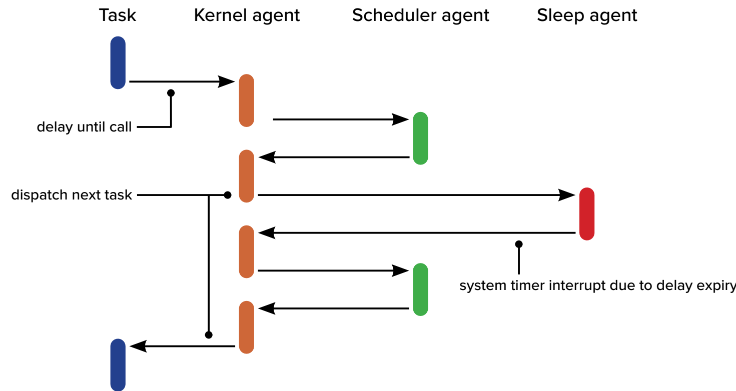
GNAT-BB was designed to support Ravenscar Profile applications on bare board systems, leading the executive to tightly integrate the FIFO_Within_Priorities task dispatching policy into its kernel. The design of Acton on the other hand grew from a desire to support different task dispatching policies and execution servers. As a result, Acton decouples scheduling from its kernel using scheduler agents: special tasks managing their own set of task queues.

By using a well-defined message interface between scheduler agent and kernel, Acton can use different task dispatching policies without having to touch Oak; enabling schedulers to be built, tested and verified separately from the kernel. This design enables the kernel and schedulers to manage their own memory demands, and the executive can individually track their execution times. Acton's approach also removes the need to use pointers within Oak to manage different task dispatching policies.

However, moving the scheduling role outside the kernel causes scheduling operations to become more expensive than those on GNAT-BB, as shown in Figure 46. This results from scheduling operations requiring context switches to and from the scheduler agent to perform the required operation, which also invokes an additional iteration of the *Oak Run-Loop*. For example, Figure 44 and Figure 45 demonstrate the context switches and priority changes required to execute a **delay until** statement and resume running after the delay expires. Acton requires eight context switches to achieve this while GNAT-BB requires only two partial context switches to handle an alarm interrupt.

Acton and GNAT-BB also take different approaches for their implementation of the FIFO_Within_Priorities task dispatching policy. GNAT-BB employs two linked list queues: a priority-ordered first-in first-out queue for all runnable tasks and a wake-time ordered queue for sleeping tasks. Both queues exhibit the same timing complexity: task removal is $O(1)$ because the kernel only removes the queue head, while $O(n)$ bounds insertions — where n is the number

Acton takes multiple context switches to processes a delay until statement.
Context view



While GNAT-BB performs the statement within the same context.
Context view. Resuming after idle: same task (left), different task (right)

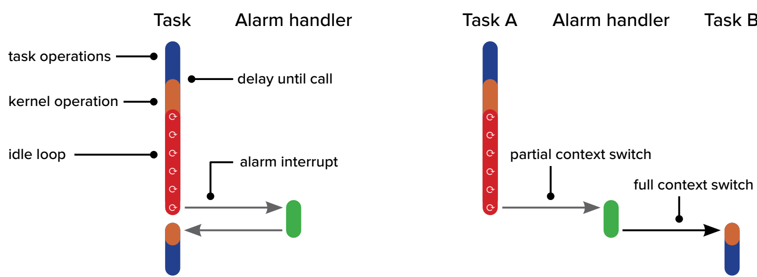


Figure 45
Context view for a task calling a delay until statement and then waking after a short sleep. Also included is the context view for GNAT-BB when a different task wakes first.

If the wake time had passed by the time the kernel handles the statement, Acton would have dispatched the task instead of the sleep agent. Likewise, GNAT-BB would have skipped the idle loop.

Resulting in GNAT-BB having a smaller delay until statement overhead than Acton.

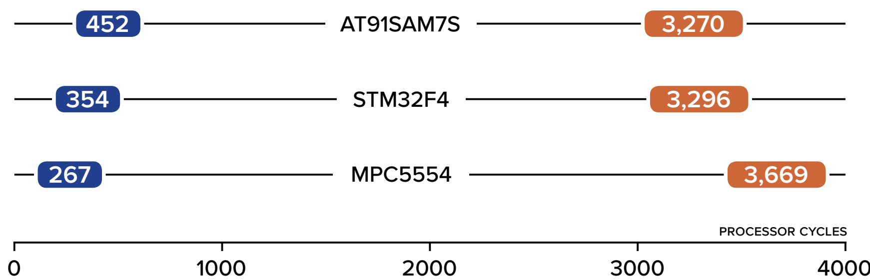


Figure 46
delay until statement overhead in cycles when given Time less than Clock.

of queued tasks. Acton similarly uses two queues for runnable and sleeping agents, but implements them using red-black trees: the runnable queue with a normal red-black tree while the sleeping queue uses a KP-RB tree. Bounding the timing complexity for insertion and removal from these trees is $O(\log n)$.

Figure 47 and Figure 49 present the execution time for two scheduling operations across the three microcontrollers. Figure 47 charts the overhead of a yield operation against the length of a queue of runnable tasks sharing the same priority. Defining this overhead is the time from the yield call (a **delay until** Ada.Real_Time.Time_First statement) to the resumption of the next runnable task, with the yield operation removing the head of the runnable queue and placing

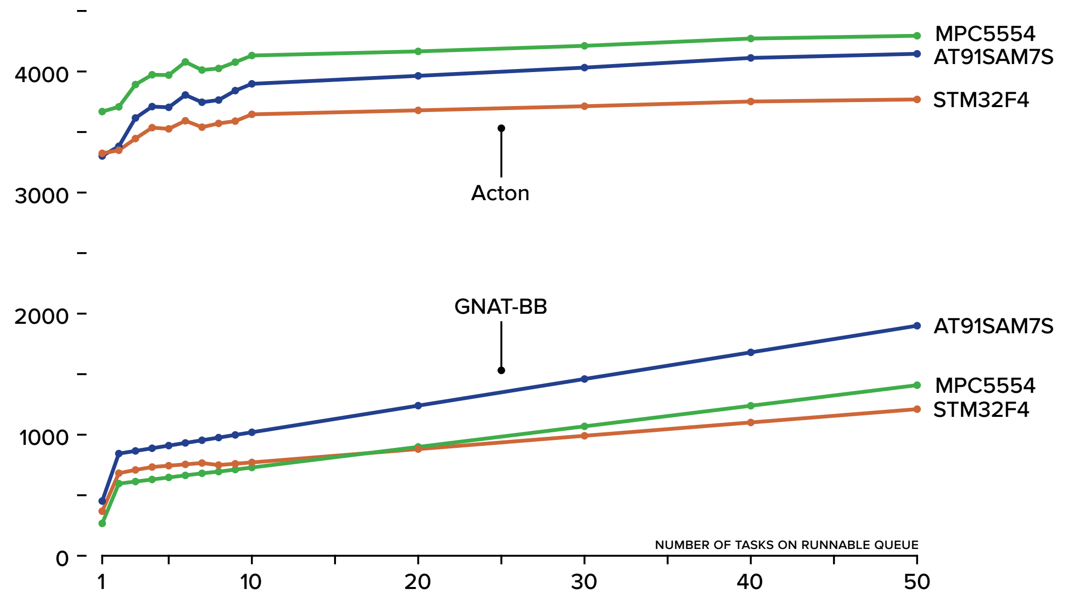


Figure 47 Upper bound for yield times against length runnable queue (time from one task yielding and the next running).

it on its tail. Visible from Figure 47 is the dominating overhead incurred by Acton due to its kernel and scheduling arrangements, and the different timing behaviours of the executives' FIFO_Within_Priorities implementations: $O(\log n)$ of Acton and $O(n)$ of GNAT-BB.

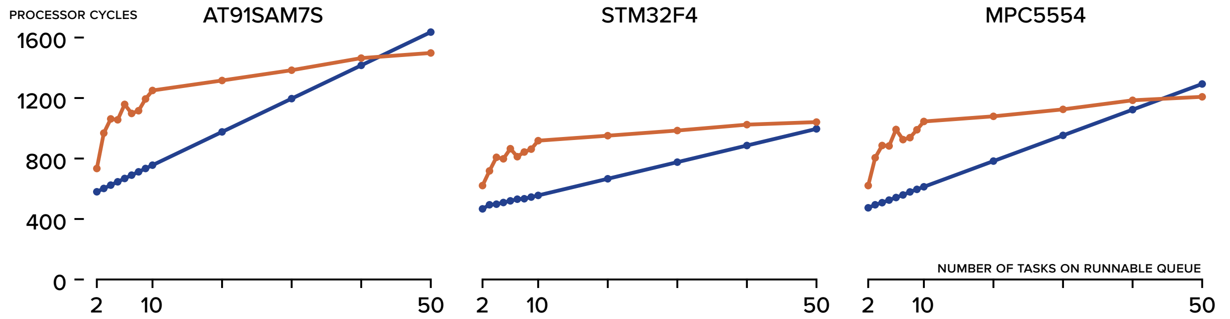
Figure 48 explores the latter by striping the kernel overhead from the results to enable a comparison between the two FIFO_Within_Priorities implementations. As shown, Acton's choice of data structure for its scheduler queues offers better scalability as the number of tasks grows: facilitating tighter bounds on the scheduling operations WCET and presents less run-time variability. For systems with a small number of tasks however (less than 40–50 tasks for the microcontrollers used here), the computation complexity of Acton's tree based queues overwhelms its scalability, with the simple linked-list queues of GNAT-BB proving more efficient.

The second scheduling operation explores the difference between the desired wake time of a task and a call to Clock just after waking, varying the number of lower priority tasks waking at the same time. Figure 49 charts the results. During the wake operation, the schedulers for both executives move all the tasks from the sleep queue to the runnable queue, testing the insertion and removal complexity of the queues under a worst-case scenario. For GNAT-BB, the wake operation scales as $O(n^2)$ while Acton scales $O(n \log n)$, where n is the number tasks. While it may be unusual for a significant number of tasks to share the same wake time, many microcontrollers like the AT91SAM7S and STM32F4 use a software-based system clock updated every 1 ms: creating a window for tasks with similar wake times to wake together.

Figure 50 takes the scheduling operation presented in Figure 49 and charts the execution time of only the queue operations. Like Figure 48, the tree-based queues of Acton provides better scalability but have larger overheads than the simple list-based queues of GNAT-BB for systems with small numbers of tasks.

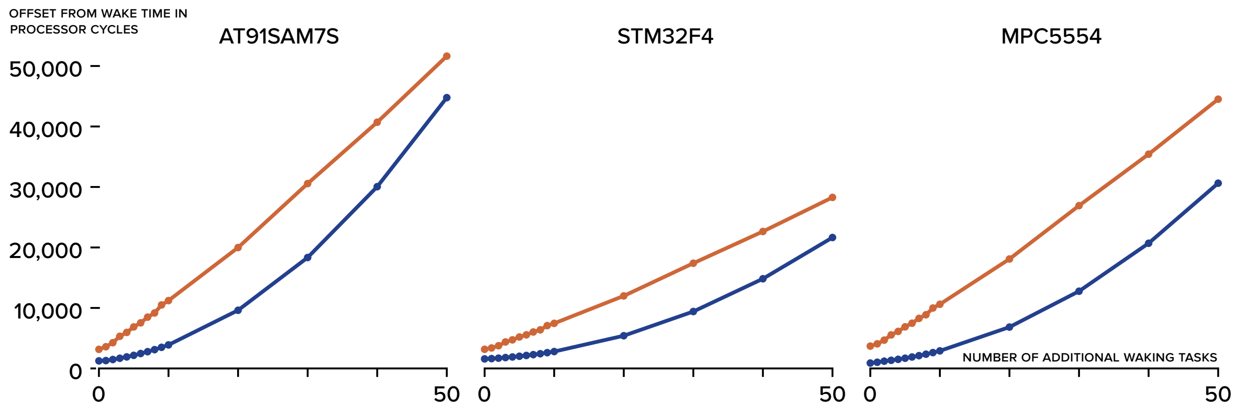
Though Acton's large yield statement overhead is not due to the queue operations the statement invokes.

Compared to GNAT-BB, Acton's operations on its queues to yield a task are less effected by the number of tasks on the runnable queue and more efficient when more than 40 tasks are present.



Acton exhibits better scaling when multiple tasks wake at the same time.

Though the operation can be considerably more expensive than GNAT-BB until the cross-over point is reached.



Though — unlike Figure 48 — Acton's queue operations when multiple tasks wake at the same time dominate the kernel overhead.

Even at low number of additional waking tasks (<10), GNAT-BB simpler queue structure is considerably more efficient.

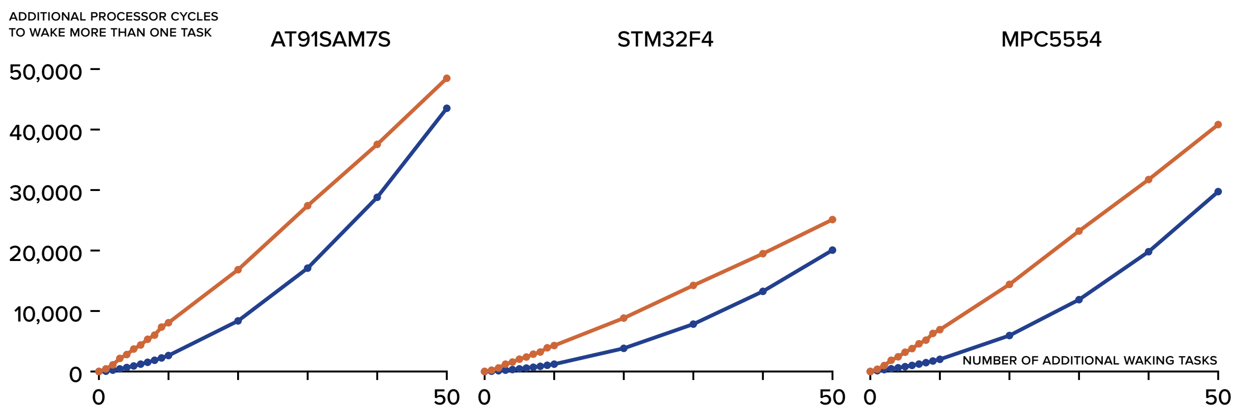


Figure 48 (top)
Execution time for the queue operations to process a yielded task.

Figure 49 (middle)
Offset from the desired wake time for a high priority task waking with addition tasks. Note the runnable queue is empty when the tasks wake.

Figure 50 (bottom)
The processor cycles required to move additional waking tasks.

However, unlike the first scheduling operation explored, the bulk movement of tasks between queues quickly dominate any kernel imposed overheads.

The choice of data structures for the queues comes from the design goals of the executives. GNAT-BB targets bare board systems typically characterised by limited task numbers and consequently the low execution costs of simple linked-list queues compensate their poor scalability. However, as Chapter 4 described, these simple queues complicate schedulability analysis because they cause lower priority tasks to pre-empt a running task when they wake and migrate to their runnable queues. Additionally, GNAT-BB assigns the execution time of this interruption to the running task, thus creating a way for tasks to consume the execution budget of a higher priority task. This interruption by lower priority tasks results from GNAT-BB selecting the first task on its alarm queue, regardless of priority.

Efficiently removing the interruption caused by lower priority tasks required Acton to choose different queue data structures. KB-RB trees became the choice for time-based queues with its ability to find the first task above a particular priority in $O(\log n)$ time. Acton uses them in two places to prevent the interruption by lower-priority tasks: as the sleeping queue in the FIFO_Within_Priority scheduler and as the timer queue for Oak. The former determines when the queues of a scheduler need servicing while the latter determines when the kernel needs to handle a raised timer. Being able to efficiently select the first task or timer above a given priority enables Acton to prevent running tasks from being interrupted by lower priority tasks.

Consequently, removing the interference caused by lower priority tasks allows Acton to simplify the timing behaviour of programs running on it, enabling these programs to use tighter WCET bounds. This comes with the compromise of higher execution overheads for small number of tasks. However, since Acton does not tie itself to a particular task dispatching policy or policy implementation, policy implementations that are better suited to the user can replace the provided scheduler agents. For example, a real-time system with a small number of tasks may use a scheduler agent implementing its runnable queue as a linked-list, allowing the use of a more efficient data structure for the runnable queue while preserving the ability for lower-priority tasks not to wake higher-priority tasks. Alternatively, GNAT-BB-like scheduler agent is usable for systems where the pre-emption problem is not an issue. Ultimately, though, the large execution time difference between scheduler operations on the two Ada executives is not down to the choice of data structures for the queues, but rather the kernel overheads imposed by placing schedulers outside the kernel.

Idle State

A separate idle thread means it takes longer for a task to resume on Acton than GNAT-BB.

Generally, task resumption on GNAT-BB is quicker when the task was the last task running as opposed to when it was a different task.

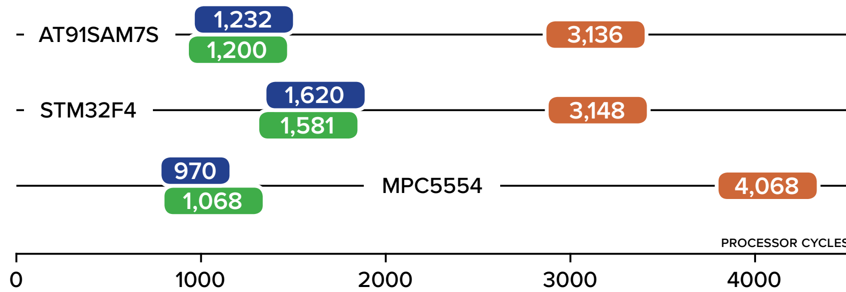


Figure 51
Offset of Ada.Real_Time.Clock from the desired wake time when called immediately after the task wakes; where it is the only task to wake

When the Ada executives have no tasks to run they enter an idle or sleep state while they wait for an interrupt to occur. Both executives implement the idle state transparently to the rest of the executive, particularly the context switching and interrupt handling code. A typical idle state implementation uses a busy-wait loop, though the loop is replaceable on many microcontrollers by low power consumption modes that halt the processor. For real-time systems spending significant time in an idle state, and where the exit overhead from the modes is acceptable, these modes help improve system energy efficiency. Of the three microcontrollers, only the STM32F4 offers a low powered mode.

Acton implements its idle state in the form of the sleep agent, a small agent dispatched when the kernel has no other agent to dispatch. Use of the sleep agent ensures Oak remains non-preemptive. In its simplest form, the agent runs at the lowest system priority with a run-loop consisting of an empty infinite loop. On microcontrollers supporting low-power modes, the loop includes a command to place the microcontroller into the desired power mode.

By contrast, the idle state implemented in GNAT-BB is a busy-wait loop within the task dispatching routine System.BB.Protection.Leave_Kernel. This loop iterates until a task becomes dispatchable, briefly enabling interrupts each iteration to permit the handling of any raised interrupts.

The idle state approach of GNAT-BB imposes lower run-time overheads: entry to the state requires no context switches and likewise on exit if kernel dispatches the previously running task. By contrast, Acton imposes a number of context switches (see Figure 44 on page 160). Figure 51 demonstrates this by showing Acton taking two to four times as long to exit the idle state.

The GNAT-BB approach also removes the tiny memory overhead an idle task imposes. However, the busy-wait loop inside the kernel means GNAT-BB cannot enter low power modes during idle because its design requires the simultaneous enabling of interrupts and entry into the mode, which no processor supports. Acton avoids this problem by using a dedicated pre-emptable agent that only executes the mode change command. Finally, the use of an agent to implement the idle state enables Acton to the track the idle time of the system using its existing facilities.

Protected Objects

Acton's kernel operation overhead dominates a protected procedure's overhead compared to GNAT-BB.
Resulting from the context switch to the kernel to enter and exit the object.

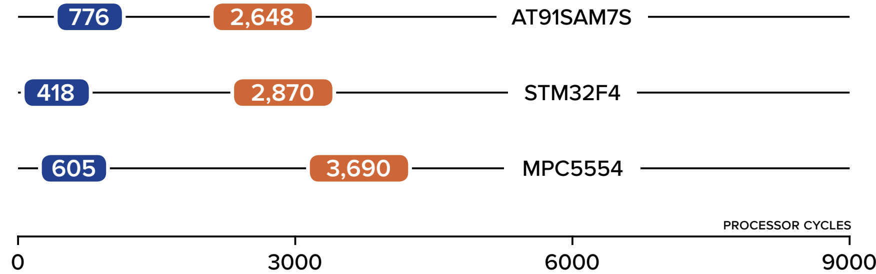


Figure 52
Overhead to access protected object without contention

Which grows when a protected action services a queued entry call.
Because Acton invokes a scheduler agent, while GNAT-BB remains in the same context.

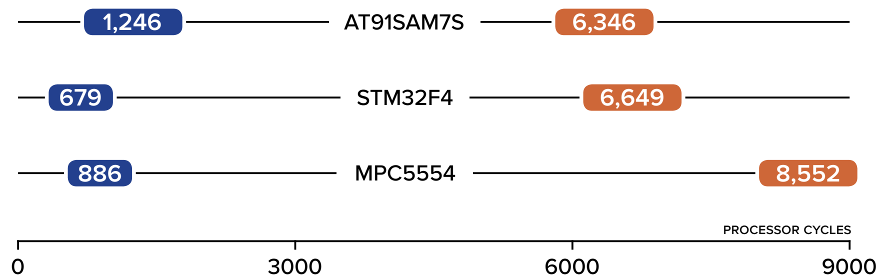


Figure 53
Execution overhead of a protected action including the servicing of a trivial entry call

Acton, by not limiting itself to the Ravenscar Profile, includes more of Ada's protected object functionality than GNAT-BB; supporting: protected objects with more than one entry, arbitrary length entry queues and non-simple entry barriers. The treatment of protected entries also differs with tasks always executing their own entry calls instead of the proxy-service approach of GNAT-BB. Consequently, real-time systems can utilise the additional protected object functionality because the self-servicing of entry calls on Acton is deterministic. This determinism results from tasks executing their own calls and the ability of Acton to transfer protected object access between tasks.

Figure 52 documents the overhead of a protected procedure call without contention on Acton and GNAT-BB. While both executives perform kernel calls during protected object entry and exit, the overhead of invoking kernel operations on Acton makes entering and exiting more expensive than on GNAT-BB. Furthermore, as the simple protected procedure call involving the servicing of an entry call in Figure 53 shows, Acton's self-service model imposes even higher overheads. This results from Acton requiring an additional kernel call to transfer the protected object to the task performing the entry call and the need to add the latter task back to its scheduler agent. Under GNAT-BB, there is no kernel call or context switch since the task performing the protected procedure services the entry call itself.

Despite the greater execution overhead on Acton, its entry call approach simplifies the timing analysis for individual tasks because a task will only perform its own operations. Thus, unlike GNAT, the timing analysis of a task concerns

only its own actions. Accordingly, this approach also removes the ability for tasks to consume the execution budget of another task. Moreover, switching from GNAT's proxy-service model to Acton's self-service approach leads to better memory and processor utilisation. A task no longer has to consider all the entries of a protected object when determining its stack size. Similarly, better processor utilisation occurs because a task's WCET does not have to consider the worst-case scenario under the proxy-service model: having to service a full set of queued entry calls after every protected operation.

Finally, the number of protected brokers allocated during the compilation of the Oak determines the number protected objects Acton can support. Analogous to the allocation of task resources, Acton's approach removes the use of memory pointers at the expense of complicating application development.

Interrupt Handling

The overhead of a protected handler invoked by an interrupt on Acton is similar to the overhead of protected procedure call.
While GNAT-BB has a notable jump in overhead since context switching and execution time tracking is invoked.

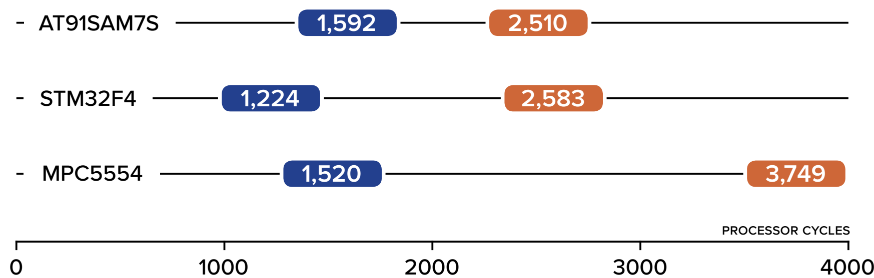


Figure 54 Overhead of invoking a protected procedure handler via an interrupt

An interrupt agent in Acton represents one interrupt priority level: providing the execution and memory resources for interrupts running at that priority. Since the dispatching of agents occurs through Oak, a raised interrupt on Acton requires a context switch to Oak before and after executing the interrupt handler. While involving more overhead than GNAT-BB, as Figure 54 shows, Acton's approach enables Oak to use its existing execution time tracking to account the execution time of an interrupt agent seperately from the interrupted task. GNAT-BB, on the other hand, tracks the execution time for each interrupt. Finally, Acton's approach sees the overhead to access a protected object similar regardless of whether the access was via a protected call or an interrupt.

Acton's interrupt handling approach assigns one stack per interrupt priority; with each interrupt stack the same size. GNAT-BB by contrast marks interrupt stack assignment as a platform specific: the STM32F4 implementation uses the same stack arrangement as Acton while the MPC5554 implementation provides a single interrupt stack shared by all interrupt priorities. The AT91SAM7S port takes a different approach again and uses the stack of the interrupted task instead of a dedicate interrupt stack. To this end, GNAT-BB provides developers more flexibility on how they assign interrupt stacks.

Memory Footprint

Figure 55 and Figure 56 document the Flash and RAM allocation for a trivial two-task program using GNAT-BB and Acton. The examples use two tasks because GNAT does not include the tasking run-time (and consequently GNAT-BB) if an application uses only one task. Both tasks consist of null-loops, with the main task having a 4 KB usable stack while the second task has 1 KB usable stack. On GNAT-BB, a usable 1 KB stack means the task has a 1.14KB task stack assigned to accommodate the overhead of the kernel (using the kernel stack overhead results from Figure 40 on page 157). Figure 56 separates interrupt stacks from the "other stacks" bar because of the different interrupt stack approaches taken by the executives and the significant contribution they make. In addition, GNAT-BB typically allocates stacks at compile time, forming part of the zero-initialised section (BSS section) while Acton currently allocates stacks at run-time. For comparison with Acton, Figure 56 separates GNAT-BB's stacks from the rest of the BSS data.

Across the three platforms, Acton has a higher memory footprint, with Figure 57 presenting theses increases relative to the footprint of GNAT-BB. Given the higher complexity of Acton this is an expected result, particularly Acton's Flash requirements: 1.5 to 3 times larger than the footprint of GNAT-BB. Dissembling the trivial application reveals a significant portion of the executive code (Figure 58) belongs to the three red-black trees used by Acton: the Oak Timer queue within the kernel, and the sleep and runnable queues used by the FIFO_Within_Priorities scheduler.

By comparison, the RAM requirements of Acton are closer to the requirements of GNAT-BB: requiring from 2.5% to 25% more RAM, excluding interrupt handlers. This result confirms Acton's complexity lies in execution overhead rather than RAM requirements. Also note that while some of some the variability between microcontrollers lies with different porting implementations — for example Acton's MPC5554 port includes routines to write to Flash — the bulk of the variability results from how the GNAT compiler translates the executives to the processor architectures.

While Acton has a larger memory footprint than GNAT-BB, the overhead for task-related objects is a different situation, as shown in Figure 59 and Figure 60. Despite the differing implementation approaches, both executives have the same run-time overhead for protected objects with entries. For protected objects without entries, GNAT-BB provides a separate implementation enabling it to reduce its run-time memory overhead for these objects; differing from Acton which only provides a sole protected object implementation.

A different situation exists for task objects, with Acton providing a smaller RAM overhead across the three different microcontrollers. Some of the difference lies with each GNAT-BB task reserving a section of their stack space for the kernel. The remaining differences, however, results from the different types of information each executive stores and how the compiler packs that information on a particular microcontroller. The ability of Acton to offer task RAM overheads competitive with GNAT-BB — despite each task having seven additional 64-bit Time variables to track the cyclic behaviour of a task — results from Acton using smaller 8-bit Agent_ID instead of 32-bit memory pointers to reference kernel objects.

Acton has significantly larger object-code stored in Flash for a trivial two task application than GNAT-BB.
While storing less data in Flash.

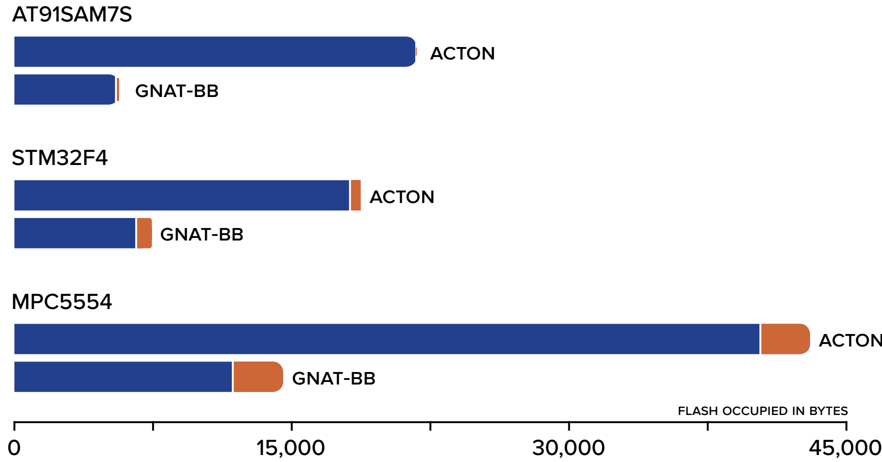


Figure 55
Allocation of Flash memory for a trivial two-task application.

While the RAM occupied by loaded data (data section), zero-initialised data (BBS section), interrupt stacks and other stacks is comparable between the executives.

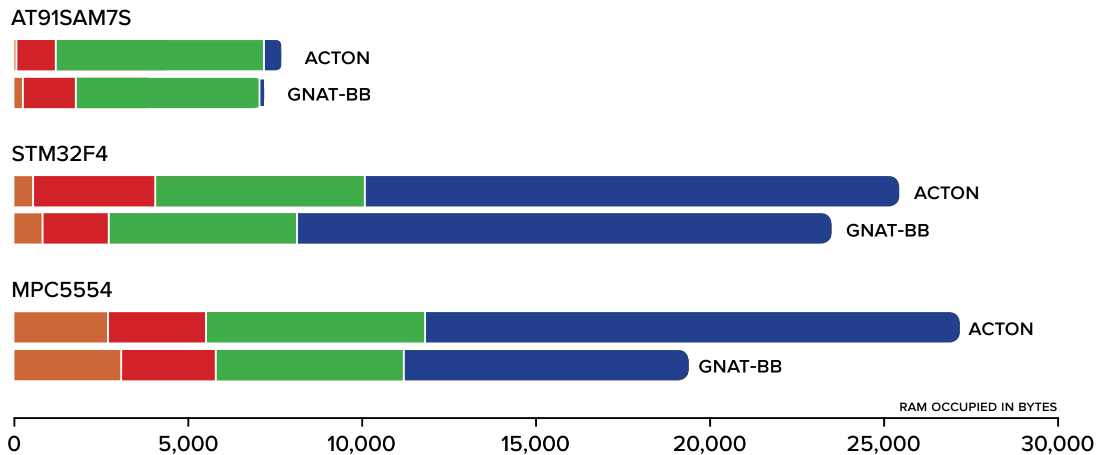


Figure 56
Allocation of RAM for a trivial two-task application. Since interrupt stacks form a large part of the space allocated for stacks, they have been separated into their own category.

As demonstrated by comparing the size of Acton's memory footprint relative to GNAT-BB's Flash and RAM footprints.

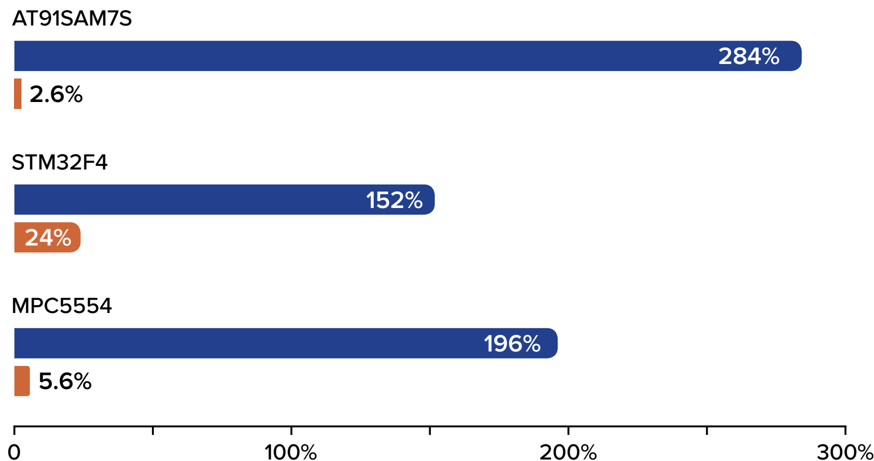


Figure 57
Acton's memory requirements relative to GNAT-BB for a trivial two-task application, excluding the interrupt stack

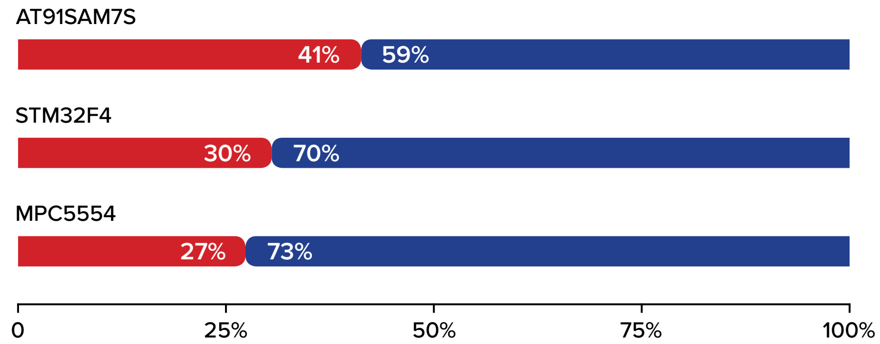


Figure 58
The percentage of Acton's object code devoted to red-black trees.

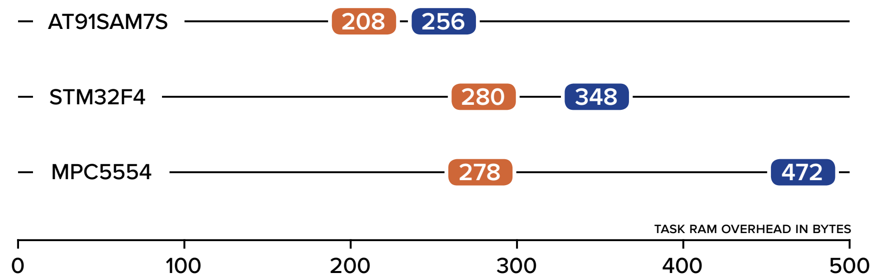


Figure 59
Run-time RAM overhead for tasks.

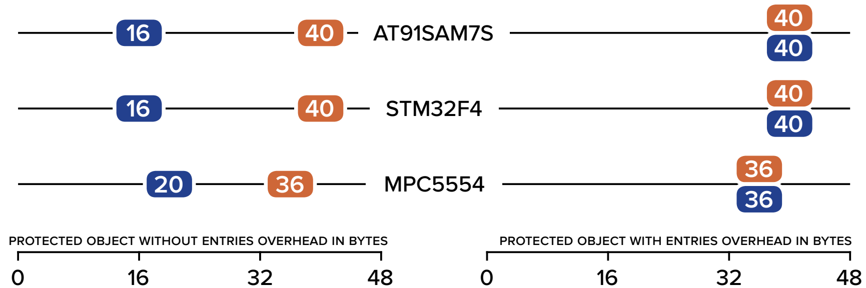


Figure 60
Run-time RAM overhead for protected objects: without entries (left), with entries (right).

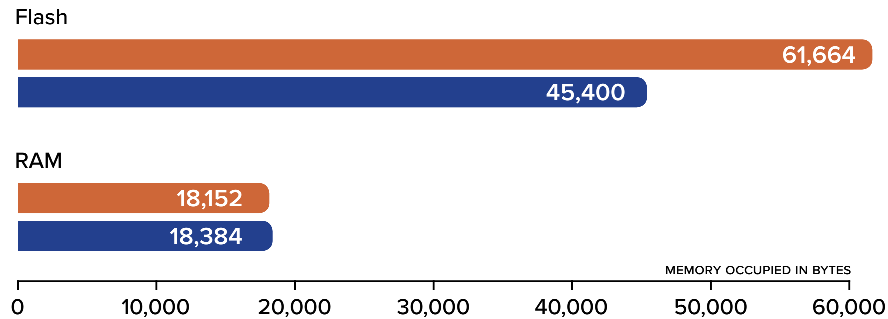


Figure 61
Memory footprint for Frank.

The Ada executives demonstrate their memory characteristics for non-trivial applications as well. Figure 61 presents the Flash and RAM requirements for Frank when using Acton and GNAT-BB. Frank consists of eight tasks and eleven protected objects. On Acton, Frank uses cyclic tasks and enforces deadlines via a common handler, while Frank on GNAT-BB does not. As shown in Figure 61, Acton uses marginally less RAM despite offering more functionality. Conversely, the 16 KB larger Flash footprint of Acton from the trivial two task application remains but does not grow.

Portability

Both Ada executives support a range of different microcontrollers based on a variety of Power Architecture and ARM cores. Additionally, GNAT-BB also runs on a number of SPARC platforms. For Acton, an early development version even ran on an 8-bit Atmel AVR microcontroller — though abandon due to the unsuitability of AVRs for real-time applications.

Acton and GNAT-BB achieve their portability by attempting to capture micro-controller specific behaviour within dedicated packages. Figure 9 on page 82 captures the approach of GNAT-BB, placing microcontroller specific operations within four dedicated packages. However, some GNAT-BB ports modify other kernel packages when the default implementation is not suitable for the target: complicating the porting processes. By contrast, Oak clearly identifies what packages are common to all platforms and what packages are microcontroller or processor core specific. This approach aims to simplify new microcontroller ports, particularly for microcontrollers sharing the same processor core as an existing port.

Language Cover

Both Acton and GNAT-BB support only a subset of Ada. Aside from restricted task support, the executives do not support Ada features requiring dynamic run-time semantics: for example heap allocation and exception propagation. The task support provided by GNAT-BB conforms to the Ravenscar Profile while Acton loosens the profile's restrictions and allows protected objects to have multiple entries with arbitrary length queues using non-simple entry barriers.

For Ada's optional Annexes, Acton and GNAT-BB offer the same ARM Annex C (*System Programming*) support as shown in Table 5. Much of this support lies at the compiler level and thus inherited from the GNAT compiler. Of the Annex features relying on the executives, both executives only support static interrupts (ARM C.3) using protected procedure handlers (ARM C.3.1). The executives provide limited support for ARM C.7 (*Task Information*): support for the Ada.Task_Identification package, no support for the package Ada.Task_Attributes and partial support for the Ada.Task_Termination package, taking into consideration tasks do not terminate on either executives.

Table 6 documents the differing support for ARM Annex D (*Real-Time Systems*), resulting from the different design goals of the Ada executives: GNAT-BB supporting Ravenscar Profile systems and Acton supporting real-time systems

Table 5
Acton and GNAT-BB ARM Annex C (Systems Programming) support

Section	Acton	GNAT-BB
C.1 Access to machine operations	yes	yes
C.2 Required representation support	yes	yes
C.3 Interrupt support	yes, static only	yes, static only
C.3.1 Protected procedure handlers	yes, static handlers only	yes, static handlers only
C.3.2 The package Interrupts	yes, static features only	yes, static handlers only
C.4 Preelaboration requirements	yes	yes
C.5 Pragma Discard_Names	yes	yes
C.6 Shared variable control	yes	yes
C.7 Task information		
C.7.1 The package Task_Identification	yes	yes
C.7.2 The package Task_Attributes	no	no
C.7.3 The package Task_Termination	subset	subset

utilising the *Cyclic Task Specification*. Consequently, GNAT-BB only supports one of the task dispatching policies in ARM D.2, in-line with the requirements of the Ravenscar Profile. Acton by contrast supports all four task-dispatching policies in ARM D.2.

GNAT-BB does provide support execution time (ARM D.14) and its interrupt handler extension (ARM D.14.3). Acton supports ARM D.14 but not ARM D.14.3 because the executive tracks the execution time of interrupt agents and not individual interrupt handlers (though support would be feasible to implement). Neither executives support execution timers (ARM D.14.1) or group execution time budgets (ARM D.14.2): GNAT-BB because of Ravenscar Profile limitations and Acton due to conflict with its *Cyclic Task Specification* implementation (Acton only supports one execution budget timer per task and is built around the assumption that only tasks of the same priority share execution budgets).

Of the remaining ARM Annex D features, Acton and GNAT-BB both support dynamic priorities (ARM D.5) within their kernels, but do not expose the support to the user. GNAT-BB because dynamic priorities lie outside the Ravenscar Profile, while the development of Acton has not required their exposure.[26] Similarly, there has been no need for Acton to support synchronous task control (ARM D.10) or synchronous barriers (ARM D.10.1). No limitation prevents Acton from supporting these synchronous features, with flexibility existing in their future implementation: either built on top of Acton's protected objects or implemented via new kernel broker type. Finally, GNAT-BB supports multiprocessor systems while Acton does not.

26. While dynamic priorities complicates schedulability analysis — the reason they are not permitted under the Ravenscar Profile — they still have value for real-time users to implement mode changes and to handle tasks which have missed their deadlines or exhausted their execution budgets.

Section	Acton	GNAT-BB
D.1 Task priorities	yes	yes
D.2 Priority scheduling	yes	yes
D.2.1 The Task Dispatching Model	yes	yes
D.2.2 Task Dispatching Pragmas	yes	yes
D.2.3 Preemptive Dispatching	yes	yes
D.2.4 Non-Preemptive Dispatching	**yes**	no
D.2.5 Round Robin Dispatching	**yes**	no
D.2.6 Earliest Deadline First Dispatching	**yes**	no
D.3 Priority ceiling locking	only Ceiling_Locking	only Ceiling_Locking
D.4 Entry queuing policies	**only FIFO_Queuing**	no
D.5 Dynamic priorities		
D.5.1 Dynamic priorities for tasks	not exposed	not exposed
D.5.2 Dynamic priorities for protected objects	no	no
D.6 Preemptive abort	no	no
D.7 Tasking restrictions	yes	yes
D.8 Monotonic time	yes	yes
D.9 Delay accuracy	yes	yes
D.10 Synchronous task control	no	**yes**
D.10.1 Synchronous barriers	no	no
D.11 Asynchronous task control	no	no
D.12 Other optimizations and determinism rules	yes	yes
D.13 The Ravenscar Profile	yes	yes
D.14 Execution time	no	**yes**
D.14.1 Execution time timers	no	no
D.14.2 Group execution time budgets	no	no
D.14.3 Execution time of interrupt handlers	no	**yes**
D.15 Timing events	yes	yes
D.16 Multiprocessor implementation	no	**yes**
D.16 Multiprocessor dispatching domains	no	no

Table 6
Acton and GNAT-BB ARM Annex D (Real-Time Programming) support

Conclusion

Acton focuses on the needs of real-time systems, differing from the existing Ada 2012 executive, GNAT-BB, which provides Ravenscar Profile compliant Ada programs an Ada executive built from an existing run-time. By removing the Ravenscar Profile restrictions, the new executive can offer real-time systems more flexibility to leverage different task dispatching policies, more protected object features and the *Cyclic Task Specification*.

Starting from a new design enables Acton to free itself from the constraints GNAT-BB encountered by using a run-time not designed for real-time or bare-board systems. However, an unexpected consequence of this freedom has been the larger execution and Flash overheads for Acton. In fact, the chapter has demonstrated that combining a modified GNAT tasking run-time with a custom kernel does produce a small and efficient Ada executive — a credit to the versatility and flexibility of GNAT. Consequentially, GNAT-BB is ideal for small Ada systems with constrained Flash and execution resources.

However, much of Acton's overheads, 10 – 30 KB extra Flash memory and 2 – 10 times larger kernel call overheads, results from addressing the deficiencies identified with GNAT-BB in Chapter 4. Predominately, the real-time issues centre on the tracking of execution time and interruptions by lower priority tasks.

For example, a significant portion of Acton's object code is devoted to the red-black trees used to implement many kernel and scheduler queues. These trees allow Acton to efficiently select timers to prevent lower priority tasks from interrupting running tasks: simplifying system schedulability analysis by removing this interruption from consideration. The trees also enable Acton to scale better as the number of tasks grows and the execution time for insertion operations is less variable than linked-lists for a given task set: allowing close alignment of WCET and average execution time figures.

The biggest contributor to Acton's execution overhead comes from placing the kernel in its own thread: forcing context switches upon all kernel operations, unlike GNAT-BB which only switches context at task dispatching points and to handle interrupts. However, accepting higher execution overheads reduces the RAM footprint of the kernel, as tasks no longer have to accommodate the kernel within their stack. Figure 59 on page 170 showed this effect with Acton having lower RAM overheads for task objects than GNAT-BB; latter confirmed with the eight-task robot software consuming less RAM on Acton than GNAT-BB (Figure 61 on page 170). Moreover, Acton's approach permits future use of memory management units to enforce hardware-level memory protection between tasks.

Furthermore, context switching is only half the execution overhead for a kernel call: tracking agent execution time forms the other. This impact was strikingly demonstrated with GNAT-BB through its ability to enable and disable execution time tracking. Enabling tracking added approximately 400 cycles to GNAT-BB's interrupt handling overhead regardless of the microcontroller (Figure 43 on page 159). Reflecting these results onto Acton, the executive spends up to a third of its interrupt handling overhead tracking the execution time of its agents.

Despite this cost, tracking the execution time of the kernel simplifies the measurement and enforcement of execution budgets. Under execution time tracking on GNAT-BB, the time for a task includes the execution of the task, its kernel calls,

the time spent moving woken tasks to the runnable queues and servicing the entry calls of other tasks. A consequence of the latter three inclusions means the execution budget of a task has to consider how other tasks may consume its budget or else the task will exhaust its budget sooner than expected.

Acton avoids this problem by tracking the execution time of its kernel separately from other agents and by implementing a self-service model for the servicing entry calls. Ergo, the execution time of a task includes only its direct activities (excluding kernel calls), making its execution time and budget independent of other tasks. As a result, maintaining a system becomes simpler since the execution budget of a task remains unaffected by modification elsewhere in the system; for example changes to other tasks or the addition of new tasks.

In addition to the cost of tracking the execution time of the kernel, there is a cost to implement the self-service model: higher execution overheads due to the additional context switches between tasks and the need to invoke a scheduler agent. In practice, the execution overhead of a protected action that services a queued entry call (Figure 53 on page 166) is double the overhead of a protected action that does not (Figure 52 on page 166).

Which leads to the questionable part of Acton's design: the decision to place schedulers in their own context. This decision makes scheduler-related kernel operations expensive since they require a context switch to a scheduler agent, invoking multiple context switches to the kernel in the process. As noted, context switches are expensive on Acton because of the context change itself and the resulting execution time updates. The result means the overhead of kernel operations on Acton can be an order of magnitude larger than those on GNAT-BB: as shown by the overheads of the delay until statement (Figure 46 on page 161) and of a protected action when servicing an entry call (Figure 53 on page 166).

Action does derive benefits from its scheduling architecture despite the impact on its kernel operations. Not tied to a single task dispatching policy, schedulers are easily swappable, even to different implementations of the same policy. Moreover, such changes have no impact on the kernel itself, removing the risk of introducing defects into the kernel, and Acton can track the execution time of schedulers separately from the kernel. Finally, Acton's scheduling approach underlies the implementation of execution servers and helps remove pointers from the kernel. While these arguments make a case for placing schedulers in their own context, unlike the rationale for a dedicated kernel context, it is debatable whether they are worth the execution overhead: particularly when similar scheduler flexibility could operate inside the context of the kernel.

Altogether, while Acton and GNAT-BB both provide similar subsets of Ada, their designs make them suited to different types of systems. GNAT-BB provides an Ada executive suitable for systems constrained by limited Flash and execution resources. Acton on the other hand suits systems requiring the enforcement of cyclic task properties at run-time to ensure their correct operation or seek the convenience provided by the *Cyclic Task Specification*. It is also suited to power-constrained applications due to its support for the low-power modes found in many microcontrollers.

2506
7

Chapter 7

Conclusion

Cyclic tasks form the cornerstone of real-time systems, driven by the nature of the fundamental system they are apart of: control systems. Since real-time systems are ultimately periodic in nature, cyclic tasks enable users to guarantee whether their system will meet its timing constraints. This is achieved by being able to perform a schedulability analysis offline, enabling users to identify timing faults early in a system's development and while the system is running; providing the necessary assurances for safety-critical systems.

However, the dissertation has shown a gap exists between the task abstraction used by real-time systems and the one provided by Ada and its environments. Ada only provides a sequential task abstraction and expects users to build cyclic tasks from the low-level tasking features the language offers. While able to build all kinds of cyclic tasks, it creates considerable and repetitive work for real-time users to build what they consider a basic abstraction. In the process, Ada's prized attributes of readability and maintainability suffer with the code making the task cyclic mixing with the code from the user's design. Furthermore, while library-based approaches can abstract the effort required to build cyclic tasks, they loose Ada's task abstraction in the process: a critical loss as it provides a visibility scope to protect the contents of a task.

Problems also exist for cyclic tasks on the only Ada 2012 environment providing an Ada executive for bare-board platforms: GNAT for Bare Boards. While

GNAT-BB provides real-time systems a small and efficient Ada executive based on a proven Ada run-time, it possesses a design impeding the timing analysis of tasks and schedulability analysis of systems. Contributing factors include flaws in tracking the execution time of tasks and the interruption of tasks by lower priority tasks. Additionally, GNAT-BB only supports one task dispatching policy and has no ability to enforce execution budgets.

Resolving this gap between the task abstraction used by real-time systems and the one provided by Ada and its environments forms the contribution of this thesis. The dissertation's first hypothesis submitted that incorporating a cyclic task abstraction into Ada's syntax and semantics would increase the reliability and maintainability of real-time systems and boost user productivity; ultimately minimising the life-cycle cost of software. The second hypothesis attended the implementation side of the abstraction, proposing a new Ada executive designed specifically for real-time systems would simplify system timing and schedulability analysis, and enable run-time enforcement of cyclic task attributes with tighter WCET bounds. Building on these hypotheses has been the *Cyclic Task Specification* and Acton: providing Ada a cyclic task abstraction and real-time executive respectively to support the claims of the hypotheses.

The *Cyclic Task Specification* supports the first hypothesis by demonstrating that Ada can elegantly incorporate a cyclic task abstraction within its existing task abstraction without the creation of new types, providing real-time systems a clarity, simplicity and structure not currently available. No longer do users have to grapple with the problems encountered by assembling cyclic tasks from low-level primitives since they can now implement cyclic tasks directly. Software reliability and maintainable improves because the *Specification* reduces the introduction of faults when cyclic tasks are written, interpreted and maintained. Productivity enhancements occur with users spending less time reading, writing, maintaining and debugging tasks. Together, these benefits enable the *Specification* to reduce the life-cycle cost of software.

Enforcing static task attributes affords users greater benefits: simplifying the discovery and interpretation of task attributes by their forced listing on the declaration of task types. Users no longer need to parse the body of a task to determine its cyclic attributes, improving information hiding since other users only need the definition of the task to discover these details. Moreover, the Ada environment ensures these attributes cannot change at run-time, aiding the timing guarantees required by real-time systems.

Crucially, code analysis tools can take advantage of these benefits too. By defining the static attributes of a cyclic task in a standardised approach, software tools utilising ASIS can now read and write the attributes straight from the code. Tools like schedulability and WCET analysers can use the same semantic information as the compiler, removing an avenue for errors occurring when copying data between code and analyser. The opportunity opens up for novel real-time compiler toolchains incorporating automatic WCET and schedulability analysis. WCET analysers can now write their results into a system's code and have these bounds enforced at run-time. Similarly, automatic schedulability analysis enables users to seamlessly detect and address timing problems as they develop their system, boosting the reliability of real-time systems.

Mitigating the complexity the *Cyclic Task Specification* adds to Ada, only a core subset integrates into the core language of Ada. This core provides the cyclic

task syntax and core cyclic task model: attractive to a wide range of application domains outside the real-time community which also use similar cyclic task patterns. The real-time element of the cyclic task model integrates with the *Real-Time Annex*, allowing Ada environments to only implement the real-time parts of the *Specification* their run-time can support, if at all.

This ability for the *Cyclic Task Specification* to split its tasking model into core and real-time components, together with implementing cyclic tasks within Ada's existing task type, addresses the existing community the aversion to implementing high-level abstractions into Ada. For all users, the *Cyclic Task Specification* provides a simple and flexible high-level abstraction: where a cyclic task is simply a task which executes its cyclic code in response to a cycle event. The *Specification* reflects this abstraction in its syntax, which only provides a way to define the initialisation and cyclic instructions of a task within Ada's existing task body syntax, plus offering a means for the user to raise cycle events. Its model then, at its simplest, provides the primitive cyclic semantics for the task: linking the rising of a cycle event with the release of a new task cycle. The colour of the model provided by the *Specification* results from the task attributes defining the source of cycle events and constraints on a cycle.

It is this definition of the source of cyclic events and cycle constraints by task attributes, and their separation from the syntax and primitive cycle semantics, that provides the power and flexibility of the *Specification*. This framework permits the development of a wide range of cyclic tasks with different cyclic constraints by task attributes alone. Demonstrating this flexibility, the core cyclic task model defines two basic methods for raising cycle events, while real-time systems have access to a new set of cycle events and cycle constraints via new task attributes to produce the wide set of cyclic tasks and cycle enforcement measures real-time systems require. Since Ada supports implementation-defined attributes, scope exists for Ada environments to extend the cyclic task model to support other cyclic tasks or cycle enforcement methods outside the language by providing the required run-time changes and supporting attributes.

Underlining the strength of the cyclic task abstraction is its recognition that cyclic tasks consist of two operation phases: initialisation and cyclic execution. Separating the task into these two sections overcomes the obstacles of trying to implement initialisation within a purely cyclic task, further addressing the concerns about the inflexibility of high-level abstractions and going beyond the offerings of other languages. The implementation of this separation also aids software reliability and readability: the two sections are part of the same task body and they share their state, simplifying finding the code of a task and reference its state. Powerfully, if the system supports dynamic attributes, the initialisation section enables the task to set its own cyclic task attributes before commencing cyclic execution; even delay its execution.

Furthermore, there are no dependencies on task dispatching policies unlike current approaches: permitting cyclic tasks to move dynamically between task dispatching policies at run-time and allowing more systems reuse unmodified tasks. Thus, not only does the *Cyclic Task Specification* supports the dissertation's first hypothesis by reducing the life-cycle cost of software, the high-level abstraction does not restrict what users can do and provides more flexibility than what is currently achievable. Even if the *Specification* does not offer the required cyclic task behaviour, an Ada environment can add the required support without language modification.

These characteristics of the *Cyclic Task Specification* separate it from approaches taken by other languages. Unlike earlier syntax-based approaches, the *Specification's* novel use of attributes and a core cyclic task model allows the definition of a wide range of cyclic task behaviours with only the introduction of a single new keyword. Moreover, new cyclic task behaviours can be implemented without affecting existing users. Compared to library-based approaches, the *Specification* enables users to directly use the cyclic task abstraction in their software and permits the static declaration and enforcement of cyclic task properties.

The dissertation has also shown the *Cyclic Task Specification* is implementable through Acton. Further, Acton supports the second hypothesis of the dissertation: showing a new Ada real-time executive does simplify the timing analysis of tasks and schedulability analysis of systems compared to GNAT-BB, and permits run-time enforcement of cyclic task attributes with tighter WCET bounds.

Acton achieves this by employing a more sophisticated scheduler and timer management approach than GNAT-BB, preventing the interruption of tasks by lower priority tasks. These changes, plus the separate execution time tracking of the kernel and a switch to a self-service model for entry calls, mean the execution time reported for a task includes only the activities of the task. Doing so enables tighter WCET bounds for the task because the WCET does not have to include any potential execution time spent doing non-task related activities. Furthermore, the WCET of a task on Acton does not change in response to changes elsewhere in the system.

The cost of Acton's approach is higher Flash and execution overhead. Much of the execution overhead results from the decision to place schedulers outside the kernel, while the remaining overhead follows from resolving the real-time impediments identified in GNAT-BB. Consequently, GNAT-BB remains the executive of choice for applications requiring low Flash and execution overheads, or uses multiple processors. On the other hand, Acton suits real-time applications seeking the advantages provided by the *Cyclic Task Specification* and the run-time enforcement of timing constraints. Acton is also more suited to power-constrained applications due to its support for processor low-power modes and for applications wishing to take advantage of its wider Ada support, including different task dispatching policies and protected objects with multiple entries.

Ultimately, underpinning the dissertation is a desire to reduce the existing life-cycle cost of real-time software by simplifying the effort to implement and sustain cyclic tasks in a real-time environment. Thus, the dissertation has made its contribution by resolving the mismatch between the cyclic task abstraction at the heart of real-time systems and the task abstraction provided by Ada and its environments. Ada will become a de-facto choice for real-time systems as it will be the only language focused on the development of reliable and maintainable systems that provides a native cyclic task abstraction matching the one used in the design of real-time systems. Consequently, users will spend less time building, debugging and maintaining these systems — ultimately reducing the life-cycle cost.

Furthermore, the value of the abstraction extends beyond real-time systems to other application domains that take advantage of cyclic task patterns. Even if the presented model does not provide the desired cyclic task behaviour, its flexibility permits the desired extension: either within the language or without.

Future Work

Unlike cyclic task abstractions provided by library-based solutions, the *Cyclic Task Specification* faces a high bar before Ada practitioners can take advantage of it: requiring the incorporation into the Ada standard. Hence, the *Specification*'s future work will take place in both the Ada real-time community and the general Ada community to develop, refine and gather acceptance for the incorporation of the *Cyclic Task Specification* into the Ada standard. The path the *Specification* takes rests on its technical merits and the views of the Ada community, particular with its introduction of a new keyword and task body syntax affecting the core language.

To help push the cause for the incorporation of the *Cyclic Task Specification* into Ada, support for the *Specification* needs to be more widely available. This entails incorporating the *Specification* into GNAT since this is the only Ada environment supporting Ada 2012's aspects. In a way, this work has already started since Acton incorporates the syntax changes into its branch of the GNAT compiler.

Acton has many paths available for it going forward. As a first step, the question of its scheduling approach needs addressing. While providing flexibility to switch and nest schedulers, it comes with a high run-time overhead. Operating scheduler agents within the kernel's context would reduce run-time overheads at the expense of this flexibility. Thus, Acton needs careful consideration going forward on the best place to locate scheduler agents that offers an acceptable compromise.

Acton should also move from Ada to Spark 2014 as the implementation language. A move to Spark would allow static verification of the kernel to enforce its correct functionally, providing the necessary assurances for high-integrity systems. It would also present a unique opportunity to provide a formally verified Ada tasking run-time. To this end, the design of Acton should simplify this transition because the kernel avoids, with few exceptions, the Ada language features restricted by Spark: in particular, the kernel uses very few access types.

Finally, Acton has the flexibility to grow the subset of Ada features it supports or shrink to provide novel subsets. New features like dynamic memory support, exception handling and task rendezvous would make Acton more attractive to soft and non real-time systems. Alternatively, as suggested in Chapter 5, stripping Acton of the concept of time creates a derivative suitable for event-response systems. An *Event Acton* variant would find use in tiny microcontrollers like Atmel's 8-bit AVRs which do not possess the necessary monotonic time source to support real-time applications.

A

Appendix A

Cyclic Task Libraries

This appendix documents the package specifications for the libraries that are apart of the *Cyclic Task Specification*. The *Specification* defines five packages:

- Ada.Cyclic_Tasks
- Ada.Cyclic_Tasks.Dynamics
- Ada.Execution_Servers
- Ada.Execution_Servers.Dynamics
- Ada.Execution_Servers.User_Servers

The separation of the procedures enabling dynamic task attributes into their own packages facilitates the use of Ada's dependency restrictions (ARM 13.12.1) by users or Ada environments to enforce static task attributes. Also note the libraries do not offer task attribute queries as attribute references provide this functionality.

Ada.Cyclic_Tasks with Preelaborate

```
with Ada.Task_Identification;
with Ada.Real_Time;

package Ada.Cyclic_Tasks with Preelaborate is

   type Task_Kind is (Yielding, Periodic, Aperiodic, Sporadic, False);

   type Early_Event_Responses is (Raise_Exception, Delay, Ignore);

   type Violation_Response is (Ignore, Handler, Abort_Cycle, Abort_Task);

   type Response_Handler is
     access protected procedure (T : Ada.Task_Indentification.Task_Id);

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

private
   ... -- not specified by the Cyclic Task Specification
end Ada.Cyclic_Tasks;
```

Ada.Cyclic_Tasks.Dynamics

```
package Ada.Cyclic_Tasks.Dynamics with Preelaborate is

   procedure Set_Cycle_Phase
     (Phase : in Ada.Real_Time.Time_Span;
      T     : in Ada.Task_Identification.Task_Id :=
                 Ada.Task_Identification.Current_Task);

   procedure Set_Cycle_Period
     (Period : in Ada.Real_Time.Time_Span;
      T      : in Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task);

   procedure Set_Cycle_MIT
     (MIT : in Ada.Real_Time.Time_Span;
      T   : in Ada.Task_Identification.Task_Id :=
               Ada.Task_Identification.Current_Task);

   procedure Set_Early_Event_Response
     (Response : in Early_Event_Responses;
      T        : in Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   procedure Set_Relative_Deadline
     (Deadline : in Ada.Real_Time.Time_Span;
      T        : in Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   procedure Set_Absolute_Deadline
     (Deadline : in Ada.Real_Time.Time;
      T        : in Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   procedure Extend_Current_Deadline
     (By : in Ada.Real_Time.Time_Span;
      T  : in Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task);

   procedure Set_Execution_Budget
     (Budget : in Ada.Real_Time.Time_Span;
      T      : in Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task);

   procedure Extend_Current_Budget
     (By : in Ada.Real_Time.Time_Span;
      T  : in Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task);
```

Ada.Cyclic_Tasks.Dynamics

```
procedure Set_Deadline_Response
      (Response : in Violation_Response;
       T        : in Ada.Task_Identification.Task_Id :=
                       Ada.Task_Identification.Current_Task);

   procedure Set_Budget_Response
     (Response : in Violation_Response;
      T        : in Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   procedure Set_Deadline_Handler
     (Handler : in Event_Response;
      T       : in Ada.Task_Identification.Task_Id :=
                   Ada.Task_Identification.Current_Task);

   procedure Set_Budget_Handler
     (Handler : in Event_Response;
      T       : in Ada.Task_Identification.Task_Id :=
                   Ada.Task_Identification.Current_Task);

private
   ... -- not specified by the Cyclic Task Specification
end Ada.Cyclic_Tasks.Dynamics;
```

Ada.Execution_Servers

```
with Ada.Real_Time;
with Ada.Task_Identification;
with System.Multiprocessors;

package Ada.Execution_Servers is

   type Execution_Server (<>) is abstract tagged limited private;

   function New_Execution_Server
     (Priority          : System.Any_Priority;
      Period            : Ada.Real_Time.Time_Span;
      Phase             : Ada.Real_Time.Time_Span := Ada.Real_Time.Time_Span_Zero;
      Execution_Budget  : Ada.Real_Time.Time_Span := Ada.Real_Time.Time_Span_Last;
      Relative_Deadline : Ada.Real_Time.Time_Span := Ada.Real_Time.Time_Span_Last;
      CPU               : System.Multiprocessors.CPU_Range :=
                          System.Multiprocessors.Not_A_Specific_CPU)
      return Execution_Server is abstract;

   function Get_Execution_Server
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
      return Execution_Server is abstract;

   function Task_In_Server
     (Server : Execution_Server;
      T      : Ada.Task_Identification.Task_Id :=
               Ada.Task_Identification.Current_Task) return  Boolean is abstract;

   function Get_Priority (Server : Execution_Server) return System.Any_Priority;

   function Get_Period (Server : Execution_Server) return Ada.Real_Time.Time_Span;

   function Get_Phase (Server : Execution_Server) return Ada.Real_Time.Time_Span;

   function Get_Execution_Budget (Server : Execution_Server)
     return Ada.Real_Time.Time_Span;

   function Get_Remaining_Execution_Budget (Server : Execution_Server)
     return Ada.Real_Time.Time_Span;

   function Get_Relative_Deadline (Server : Execution_Server)
     return Ada.Real_Time.Time_Span;

   function Get_Absolute_Deadline (Server : Execution_Server)
     return Ada.Real_Time.Time;

   function Get_CPU (Server : Execution_Server)
     return System.Multiprocessors.CPU_Range;
private
   ... -- not specified by the Cyclic Task Specification
end Ada.Execution_Servers;
```

Ada.Execution_Servers.Dynamics

```
package Ada.Execution_Servers.Dynamics is

   type Dynamic_Execution_Server (<>)
     is abstract new Execution_Server with abstract private;

   procedure Add_Task_To_Server
     (Server : in Dynamic_Execution_Server;
      T      : in Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task);

   procedure Remove_Task_From_Server
     (Server : in Dynamic_Execution_Server;
      T      : in Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task);

   procedure Set_Cycle_Phase (Server : in Dynamic_Execution_Server;
                              Phase  : in Ada.Real_Time.Time_Span);

   procedure Set_Cycle_Period (Server : in Dynamic_Execution_Server;
                               Period : in Ada.Real_Time.Time_Span);

   procedure Set_Relative_Deadline (Server   : in Dynamic_Execution_Server;
                                    Deadline : in Ada.Real_Time.Time_Span);

   procedure Set_Absolute_Deadline (Server   : in Dynamic_Execution_Server;
                                    Deadline : in Ada.Real_Time.Time);

   procedure Extend_Current_Deadline (Server : in Dynamic_Execution_Server;
                                      By     : in Ada.Real_Time.Time_Span);

   procedure Set_Execution_Budget (Server : in Dynamic_Execution_Server;
                                   Budget : in Ada.Real_Time.Time_Span);

   procedure Extend_Current_Budget (Server : in Dynamic_Execution_Server;
                                    By     : in Ada.Real_Time.Time_Span);

private
   ... -- not specified by the Cyclic Task Specification
end Ada.Execution_Servers.Dynamics;
```

Ada.Execution_Servers.Dynamics;

```
with Ada.Execution_Servers.Dynamics;

package Ada.Execution_Servers.User_Servers is

  type User_Execution_Server (<>)
     is abstract new Dynamics.Dynamic_Execution_Server with abstract private;

   procedure Add_Task_To_User_Server
     (Server : in User_Execution_Server;
      T      : in Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task) is abstract;

   procedure Remove_Task_From_User_Server
     (Server : in User_Execution_Server;
      T      : in Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task) is abstract;

private
   ... -- not specified by the Cyclic Task Specification
end Ada.Execution_Servers.User_Servers;
```

B

Appendix B

References

Ada 9x Project Office (1990a)
Ada 9x Project Report: Ada 9x Revision Issues Release 2
Office of the Under Secretary of Defense for Acquisition,
United States Department of Defense: Washington, DC.
DTIC: ADA223166

Ada 9x Project Office (1990b)
Ada 9x Project Revision Request Report: Supplement 1
Office of the Under Secretary of Defense for Acquisition,
United States Department of Defense: Washington, DC.
DTIC: ADA222159

Ada Runtime Environment Working Group (1991)
Catalogue of Interface Features and Options for the Ada Runtime Environment
Ada Letters: Volume XI, Issue 8. ACM: New York.

AdaCore (2013a)
GNAT for LEGO Mindstorms NXT
AdaCore: New York.
WEB: libre.adacore.com/academia/mindstorms/
retrieved August 1, 2013

AdaCore (2013b)
GNAT Reusable Components (Version 1.6w)
AdaCore: New York.
WEB: docs.adacore.com/gnatcoll-docs/
retrieved August 28, 2013.

AdaCore (2014)
ASIS-for-GNAT User's Guide (Version 17.0w)
AdaCore: New York.
WEB: docs.adacore.com/asis-docs/asis_ug.html
retrieved December 13, 2015.

AdaCore (2015a)
GNAT Reference Manual (Version 7.4.0w)
AdaCore: New York.
WEB: docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm.html
retrieved June 21, 2015.

AdaCore (2015b)
GNAT User's Guide Supplement for Cross Platforms (Version 7.4.0w)
AdaCore: New York.
WEB: docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html
retrieved June 21, 2015.

AdaCore (2015c)
GNAT User's Guide Supplement for GNAT Pro Safety-Critical and GNAT Pro High-Security (Version 7.4.0w)
AdaCore: New York.
WEB: docs.adacore.com/gnathie_ug-docs/html/gnathie_ug/gnathie_ug.html
retrieved June 21, 2015.

AdaCore (2015d)
GNAT User's Guide for Native Platforms (Version 7.4.0w)
AdaCore: New York.
WEB: docs.adacore.com/gnat_ugn-docs/html/gnat_ugn.html
retrieved June 21, 2015.

Mario Aldea, Javier Miranda & Michael González Harbour (2004)
Implementing an Application-Defined Scheduling Framework for Ada Tasking
Presented at the Ninth Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain.
Springer Berlin Heidelberg.
DOI: 10.1007/978-3-540-24841-5_23

Angel Alvarez (1987)
Real-time programming and priority interrupt systems
Presented at the First International Workshop on Real-Time Ada Issues, Moretonhampstead, UK.
ACM: New York.
DOI: 10.1145/36821.36813

Neil C. Audsley, Alan Burns, M. F. Richardson, & Andy Wellings (1991)
Hard Real-Time Scheduling: The Deadline-Monotonic Approach
Presented at the Eighth IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA.
IEEE Computer Society Press.

Ted Baker (1989)
Fixing some time-related problems in Ada
Presented at the Third International Workshop on Real-Time Ada Issues (IRTAW '89), Farmington, PA, USA.
ACM: New York.
DOI: 10.1145/91354.91382

Ted Baker & Edward W. Giering (1996)
POSIX/Ada Real-Time Bindings Final Technical Report
Technical report to the U.S. Army CECOM.
Telos Systems Group Contract.
Florida State University: Tallahassee, FL, USA.

Ted Baker, Dong-Ik Oh & Seung-Jin Moon (1997)
Low-level Ada tasking support for GNAT-performance and portability improvements
Ada Letters: Volume XVII, Issue 3. ACM: New York.
DOI: 10.1145/261374.262780

Ted Baker & Alan Shaw (1988)
The Cyclic Executive Model and Ada
Presented at the Ninth IEEE Real-Time Systems Symposium, Huntsville, AL, USA. IEEE.
DOI: 10.1109/REAL.1988.51108

John Barnes (Ed.) (1995)
Ada 95 Rationale: The Language, the Standard Libraries
Springer-Verlag Berlin Heidelberg, Intermetrics Inc.
DOI: 10.1007/BFB0051526
WEB: http://www.adaic.org/resources/add_content/standards/95RAT/RAT95HTML/RAT95-contents.html

Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull (2000)
The Real-Time Specification for Java
Addison Wesley Longman: Boston MA, USA

Geert Bosch (2013)
Lock-Free Protected Types for Real-Time Ada
Ada Letters: Volume 33, Issue 2. ACM: New York.
DOI: 10.1145/2552999.2553007

Alan Burns, Brian Dobbing & Tullio Vardanega (2004)
Guide for the use of the Ada Ravenscar Profile in high integrity systems
Ada Letters: Volume XXIV, Issue 2. ACM: New York.
DOI: 10.1145/997119.997120

Alan Burns & Andy Wellings (1988)
The Use of Ada in Hard Real-time Systems: Scheduling Requirements and Language Features.
Presented at the Eleventh Australian Australian Computer Science Conference, Brisbane, Australia.
Australian Computer Science Conference: Sydney

Alan Burns & Andy Wellings (2006)
Programming Execution-Time Servers in Ada 2005
Presented at the 27TH IEEE International Real-Time Systems Symposium, Rio de Janiero, Brazil. IEEE.
DOI: 10.1109/RTSS.2006.39

Alan Burns & Andy Wellings (1987)
Real-Time Ada Issues
Presented at the First International Workshop on Real-Time Ada Issues.
Ada Letters: Volume VII, Issue 6. ACM: New York.
DOI: 10.1145/36792.36800

Alan Burns & Andy Wellings (2009)
Real-Time Systems and Programming Languages (Fourth Edition)
Addison-Wesley Longman.
ISBN: 978-0321417459

Giorgio C. Buttazzo (2011)
Hard Real-Time Computing Systems (Thrid Edition)
Springer US: Boston MA, USA.
DOI: 10.1007/978-1-4614-0676-1

Fabien Chouteau & José F. Ruiz (2011)
Design and Implementation of a Ravenscar Extension for Multiprocessors
Presented at the 16TH Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK.
Springer Berlin Heidelberg.
DOI: 10.1007/978-3-642-21338-0_3

Dennis Cornhilll, Lui Sha, John P. Lehoczky, Ragunathan Rajkumar & Hide Tokuda (1987)
Limitations of Ada for Real-Time Scheduling
Presented at the First International Workshop on Real-Time Ada Issues, Moretonhampstead, UK.
Ada Letters: Volume VII, Issue 6. ACM: New York.
DOI: 10.1145/36792.36798

Francis Cottet, Joëlle Delacroix, Claude Kaiser, Zoubir Mammeri (2002)
Scheduling in Real-Time Systems
John Wiley & Sons: Chichester, UK
DOI: 10.1002/0470856343

Cyrille Comar, Franco Gasperoni & Edmond Schonberg (1994)
The GNAT Project: A GNU-ADA9X Compiler.
Technical report
New York University: New York.

Robert I. Davis & Alan Burns (2011)
A Survey of Hard Real-Time Scheduling for Multiprocessor Systems
ACM Computing Surveys: Volume 43, Issue 4. ACM: New York.
DOI: 10.1145/1978802.1978814

Juan A. de la Puente, José F. Ruiz & Juan Zamorano (2000A)
An Open Ravenscar Real-Time Kernel for GNAT.
Presented at the 5TH Ada-Europe International Conference on Reliable Software Technologies, Potsdam, Germany.
Springer Berlin Heidelberg.
DOI: 10.1007/10722060_4

Juan A. de la Puente, José F. Ruiz & Jesús M. González-Barahona (1999)
Real-Time Programming with GNAT: Specialised Kernels Versus POSIX Threads
Presented at the 9TH International Real-Time Ada Workshop, Wakulla Springs Lodge, Florida, USA
Ada Letters: Volume XIX, Issue 2. ACM: New York.
DOI: 10.1145/329607.334742

Juan A. de la Puente, Juan Zamorano & Jorge López (2009)
GNATforLEON/ORK+ User Manual (Version 1.3)
Technical Report
Universidad Politécnica de Madrid: Mardrid.

Juan A. de la Puent, Juan Zamorano, José F. Ruiz, Ramón Fernández & Rodrigo García (2000B)
The Design and Implementation of the Open Ravenscar Kernel
Presented at the 10TH International Real-Time Ada Workshop, Ávila, Spain.
Ada Letters: Volume XXI, Issue 1. ACM: New York
DOI : 10.1145/374369.374387

DoD (U.S. Department of Defense) (1983).
Reference Manual for the Ada Programming Language
U.S. Department of Defense: Washington, DC.
STANDARD: ANSI/MIL-STD-1815A

Robert Dewar (1994)
The GNAT Compilation Model
Presented at TRI-Ada '94, Baltimore MD, USA.
ACM: New York.
DOI: 10.1145/197694.197708

DIN (1998)
Programming language PEARL - PEARL 90
Deutsches Institut für Normung, Germany
STANDARD: DIN 66253-2:1998-04

Kevin Elphinstone & Gernot Heiser (2013)
From L3 to SEL4 What Have We Learnt in 20 Years Of L4 Microkernels?
Presented at the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farmington, PA, USA.
ACM: New York
DOI: 10.1145/2517349.2522720

David A. Fisher (1978)
DoD's Common Programming Language Effort
Computer: Volume 11, Issue 3. IEEE.
DOI: 10.1109/C-M.1978.218092

Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield & Mark Williamson (2004)
Safe Hardware Access with the Xen Virtual Machine Monitor
Presented at the First Workshop on Operating System and Architectural Support for the On-Demand IT InfraStructure, Boston, USA.

FSF (Free Software Foundation) (2014)
GCC (Version 5.0)
Software. Free Software Foundation.

GI-Working Group 4.4.2 (Ed.) (1998)
PEARL 90 Language Report, 2.2
WEB: http://www.real-time.de/misc/PEARL90-LanguageReport-V2.2-GI-1998-eng.pdf
retrieved January 15, 2017

Edward W. Giering & Ted Baker (1992)
Using POSIX Threads to Implement Ada Tasking: Description of Work in Progress
Presented at TRI-Ada '92, Orlando FL, USA.
ACM: New York.
DOI: 10.1145/143557.144009

Edward W. Giering & Ted Baker (1994)
The GNU Ada runtime library (GNARL)
Presented at Eleventh Washington Ada Symposium (WADAS '94), McLean VA, USA.
ACM: New York
DOI: 10.1145/197978.197989

Edward W. Giering, Frank Mueller & Ted Baker (1993)
Implementing Ada 9X features using POSIX Threads: Design Issues
Presented at the TRI-Ada '93, Seattle WA, USA.
ACM: New York
DOI: 10.1145/170657.170736

Edward W. Giering, Frank Mueller & Ted Baker (1994)
Features of the GNU Ada runtime library
Presented at the TRI-Ada '94, Baltimore MD, USA
ACM: New Yor
DOI: 10.1145/197694.197711

James J. Hunt, Ben Brosgol, Andy Wellings, Kelvin Nilsen, Ethan Blanton, Peter Dibble, and David Holmes (2017)
Realtime and Embedded Specification for Java, Version 2.0 (Draft 56)
aicas GmbH, Karlsruhe, Germany, .
WEB: http://aicas.com/cms/sites/default/files/rtsj_56.pdf
retrieved January 13, 2017

Jean D. Ichbiah, John Barnes, Robert J. Firth & Mike Woodger (1986)
Rationale for the Design of the Ada Programming Language
Ada Joint Program Office,
United States Department of Defense: Washington, DC.
DTIC: ADA187106

HOLWG (High Order Language Working Group) (1977)
Department of Defense Requirements for High Order Computer Programming Languages "IRONMAN" Revised
United States Department of Defense: Washington, DC.

HOLWG (High Order Language Working Group) (1978)
Department of Defense Requirements for High Order Computer Programming Languages "STEELMAN"
United States Department of Defense: Washington, DC.
DTIC: ADA059444

IEEE Computer Society (2008)
IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications (Issue 7)
IEEE: New York, USA.
STANDARD: IEEE 1003.1-2008
DOI: 10.1109/IEEESTD.2008.4694976

ISO/IEC (International Organization for Standardization/International Electrotechnical Commission) (2012)
ISO/IEC 8652:1995 Ada (Second Edition)
ISO/IEC: Switzerland.
STANDARD: ISO/IEC 8652:1995

ISO/IEC (1999)
ISO/IEC 15291:1999 Ada Semantic Interface Specification
ISO/IEC: Switzerland.
STANDARD: ISO/IEC 15291:1999

ISO/IEC (2000)
Guide for the Use of the Ada Programming Language in High Integrity Systems
Technical Report
ISO/IEC: Switzerland.
REPORT NUMBER: ISO/IEC TR 15942

ISO/IEC (2012)
ISO/IEC 8652:2012 Ada (Third Edition)
ISO/IEC: Switzerland.
STANDARD: ISO/IEC 8652:2012

Eugene Kligerman & Alexander D. Stoyenko, (1986)
Real-Time Euclid: a Language for Reliable Real-Time Systems.
IEEE Transactions on Software Engineering: Volume SE-12, Issue 9. IEEE.
DOI: 10.1109/TSE.1986.6313049

Allan Klumpp (1985)
Space Station Flight Software: Hal/S or Ada?
Computer: Volume 18, Issue 3. IEEE
DOI: 10.1109/MC.1985.1662827

Wilfried Kneis (1981)
Draft standard industrial real-time FORTRAN
ACM *SIGPLAN Notices*: Volume *16*, Issue 7. ACM, New York.
DOI: 10.1145/947864.947868

Hermann Kopetz (2011)
Real-Time Systems: Design Principles for Distributed Embedded Applications (Second Edition)
Springer: New York
DOI: 10.1007/978-1-4419-8237-7

Phillip A. Laplante & Seppo J. Ovaska (2011)
Real-Time Systems Design and Analysis: Tools for the Practitioner (Fourth Edition)
Wiley-IEEE Press: Hoboken, NJ, USA.
DOI: 10.1002/9781118136607

Insup Lee & Vijay Gehlot (1985)
Language Constructs for Distributed Real-Time Programming.
Presented at the 6TH IEEE Real-Time Systems Symposium,
San Diego CA, USA.
IEEE Computer Society Press.

John P. Lehoczky & Sandra Ramos-Thuel (1992)
An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems
Presented at the 13TH IEEE Real-Time Systems Symposium,
Phoenix AZ, USA.
IEEE Computer Society Press.
DOI: 10.1109/REAL.1992.242671

Jochen Liedtke (1995)
On μ-Kernel Construction
Presented at the 15TH ACM Symposium on Operating Systems Principles,
Copper Mountain CO, USA.
ACM: New York.
DOI: 10.1145/224057.224075

Jochen Liedtke (1996)
Toward Real Microkernels.
Communications of the ACM
ACM: New York.
DOI: 10.1145/234215.234473

C.L. Liu & James W. Layland (1973)
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment
Journal of the ACM: Volume 20, Issue 1.
DOI: 10.1145/321738.321743

Jane W.S. Liu (2000)
Real Time Systems
Prentice Hall: Englewood Cliffs NJ, USA.

G.K. Manacher (1967)
Production and Stabilization of Real-Time Task Schedules
Journal of the ACM: Volume 14, Issue 3.
ACM: New York.
DOI: 10.1145/321406.321408

Tomas Martin (1978)
Realtime programming language PEARL - Concept and characteristics
Presented at the Second International Computer Software and Applications Conference, Chicago IL, USA.
IEEE: New York.
DOI: 10.1109/CMPSAC.1978.810404

Ruth A. Maule (1986)
Run-Time Implementation Issues for Real-Time Embedded Ada
Presented at the First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, TX, USA.
University of Houston-Clear Lake: Clear Lake TX, USA.

Frank McCormick (1987)
Scheduling difficulties of Ada in the hard real-time environment.
Presented at the First International Workshop on Real-Time Ada Issues.
Ada Letters: Volume VII, Issue 6. ACM: New York.
DOI: 10.1145/36821.36802

Javier Miranda & Edmond Schonberg (2004)
GNAT: The GNU Ada Compiler
AdaCore: New York
WEB: www.adacore.com/gap-static/GNAT_Book/html/

Dennis Moralee (1981)
ADA: software engineering language of the future?
Electronics and Power: Volume 27, Issue 7-8
The Institution of Electrical Engineers: UK
DOI: 10.1049/EP.1981.0270

Philip Newbold, Arra Avakian, Carl Helmers, Andy Johnson, ROn Kole, Dan Lickly, Fred Martin, Joe Saponaro, and Woody Vandever (1976)
HAL/S Langauge Specification
Intermetrics: Cambridge MA, USA.

OAR (On-Line Applications Research Corporation) (2011)
RTEMS Applications Ada User's Guide (Version 4.10.2)
WEB: docs.rtems.org/doc-current/share/rtems/html/ada_user/index.html
retrieved December 13, 2011.

Michael J. Ryer (1979)
Programming in HAL/S (Second Edition).
Intermetrics: Cambridge MA, USA.

Mario Aldea Rivas & Michael González Harbour (2001)
MARTE OS: An Ada Kernel for Real-Time Embedded Applications
Presented at the Sixth Ada-Europe International Conference on Reliable Software Technologies, Leuven, Belgium.
Springer Berlin Heidelberg
DOI: 10.1007/3-540-45136-6_24

Mario Aldea Rivas & Michael González Harbour (2002)
Application-Defined Scheduling in Ada
Presented at the 11TH International Real-Time Ada Workshop, Mt. Tremblant, Quebec, Canada.
Ada Letters: Volume XXII, Issue 4. ACM: New York
DOI: 10.1145/584417.584429

Mario Aldea Rivas, Michael González Harbour & José F. Ruiz (2009)
Implementation of the Ada 2005 Task Dispatching Model in MARTE OS and GNAT.
Presented at the 14TH Ada-Europe International Conference, Brest, France
Springer Berlin Heidelberg.
DOI: 10.1007/978-3-642-01924-1_8

Mario Aldea Rivas, Javier Miranda & Michael González Harbour (2005)
Integrating Application-Defined Scheduling with the New Dispatching Policies for Ada Tasks
Presented at the 10TH Ada-Europe International Conference on Reliable Software Technologies, York, UK.
Springer Berlin Heidelberg.
DOI: 10.1007/11499909_18

Edmond Schonberg & Bernard Banner (1994)
The GNAT Project: A GNU-Ada 9X Compiler
Presented at the TRI-Ada '94, Baltimore, Maryland, USA.
ACM: New York.
DOI: 10.1145/197694.197706

Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Ted Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, Aloysius K. Mok (2004)
Real Time Scheduling Theory: A Historical Perspective.
Real-Time Systems: Volume 28, Issue 2/3. Kluwer Academic Publishers
DOI: 10.1023/B:TIME.0000045315.61234.1E

John A. Stankovic (1988)
Misconceptions about real-time computing: a serious problem for next-generation systems.
Computer: Volume 21, Issue 10. IEEE: New York.
DOI: 10.1109/2.7053

Alexander D. Stoyenko (1992)
The evolution and state-of-the-art of real-time languages.
Journal of Systems and Software: Volume 18, Issue 1
Elsevier Science Publishing Co: New York NY, USA.
DOI: 10.1016/0164-1212(92)90046-M

Joyce L. Tokar (1992)
An analysis of the Ada 9X and ARTEWG CIFO 3.0 overlap
Presented at Ninth Washington Ada Symposium (WADAS '92),
La Jolla CA, USA.
ACM: New York
DOI: 10.1145/257683.257691

Andy Wellings & Alan Burns (2007)
A framework for real-time utilities for Ada 2005
Presented at the 13TH International Real-Time Ada Workshop (IRTAW 2007),
Woodstock VT, USA.
Ada Letters: Volume XXVII, Issue 2. ACM: New York.
DOI: 10.1145/1316002.1316011

Andy Wellings & Alan Burns (2007)
Real-Time Utilities for Ada 2005
Presented at the 12TH Ada-Europe International Conference on Reliable
Software Technologies (Ada-Europe 2007), Geneva, Switzerland.
Springer Berlin Heidelberg.
DOI: 10.1007/978-3-540-73230-3

Andy Wellings & Joyce L. Tokar (2003)
Session: Integration versus Orthogonality (RTSJ scheduling policies versus Ada's)
Presented at the Twelfth International Real-Time Ada Workshop
(IRTAW 2003), Viana do Castelo, Portugal.
Ada Letters: Volume XXIII, Issue 4. ACM: New York.
DOI: 10.1145/959221.959224

William A. Whitaker (1993)
Ada – The Project: The DoD High Order Language Working Group.
Presented at the Second ACM SIGPLAN Conference on History of
Programming Languages, Cambridge MA, USA.
ACM: New York.
DOI: 10.1145/154766.155376

Juan Zamorano, Alejandro Alonso, José Antonio Pulido & Juan Antonio de la Puente (2004)
Implementing Execution-Time Clocks for the Ada Ravenscar Profile
Presented at the 9TH Ada-Europe International Conference on Reliable
Software Technologies (Ada-Europe 2004), Palma de Mallorca, Spain.
Springer Berlin Heidelberg.
DOI: 10.1007/978-3-540-24841-5_10