TECHNICAL REPORT

ISO/IEC TR 24718

First edition
2005-02-15

Information technology — Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems

Technologies de l'information — Langages de programmation — Guide pour l'usage de «Ada Ravenscar Profile» dans les systèmes de haute intégrité

Reference number
ISO/IEC TR 24718:2005(E)

© ISO/IEC 2005

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2005

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2005 – All rights reserved

Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24718, which is a Technical Report of type 3, was prepared by the University of York for the British Standards Institution (BSI) as guidelines published in 2003, and was adopted (without modifications) by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

© ISO/IEC 2005 All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

The use of Ada has proven to be of great value within high integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. Such subsets have been defined for Ada 83, but these have excluded tasking on the grounds of its non-determinism and inefficiency. Advances in the area of schedulability analysis currently allow hard deadlines to be checked, even in the presence of a run-time system that enforces preemptive task scheduling based on multiple priorities. This valuable research work has been mapped onto a number of new Ada constructs and rules that have been incorporated into the Real-Time Annex of the Ada language standard. This has opened the way for these tasking constructs to be used in high integrity subsets whilst retaining the core elements of predictability and reliability.

The Ravenscar Profile is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding only during system integration and testing that the design fails to meet its non-functional requirements.

The Ravenscar Profile has been designed such that the restricted form of tasking that it defines can be used even for software that needs to be verified to the very highest integrity levels. The aim of this guide is to give a complete description of the motivations behind the Profile, to show how conformant programs can be analysed and to give examples of usage.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Information technology — Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems

1 Scope

This Technical Report provides a description of the motivations behind the Ravenscar Profile, to show how Ada programs using the profile can be analysed, and gives examples of usage.

2 Recommendations

The technical recommendations are those made in the following publication (reproduced on the following pages), which is adopted as a Technical Report:

Guide for the use of the Ada Ravenscar Profile in high integrity systems, Alan Burns, Brian Dobbing, and Tullio Vardanega, University of York Technical Report YCS-2003-348, January 2003.

For the purposes of international standardization, the modifications outlined below shall apply to the specific clause and paragraphs of the University of York publication.

Page i to ii (of the University of York publication)

This is information relevant to the University of York publication only.

Page 73

Clause 9

Substitute the following for the corresponding reference

[GA] ISO/IEC TR 15942:2000, *Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems*

[RM] ISO/IEC 8652, *Information technology — Programming languages — Ada*

3 Revision of the University of York publication

It has been agreed with the University of York that ISO/IEC JTC 1/SC 22 will be consulted in the event of any revision or amendment of this University of York publication. To this end, the British Standards Institution (BSI) will act as a liaison body between the University of York and ISO/IEC.

© ISO/IEC 2005 All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Guide for the use of the Ada Ravenscar Profile in high integrity systems

Alan Burns, Brian Dobbing and Tullio Vardanega

University of York Technical Report YCS-2003-348
January 2003

© 2003 by the authors

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale
© ISO/IEC 2005 – All rights reserved

Blank page

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Contents

1 Introduction....1
Structure of the Guide....2
Readership....2
Conventions....2

2 Motivation for the Ravenscar Profile....3
2.1 Scheduling Theory....3
2.1.1 Tasks Characteristics....3
2.1.2 Scheduling Model....4
2.2 Mapping Ada to the Scheduling Model....5
2.3 Non-Preemptive Scheduling and Ravenscar....6
2.4 Other Program Verification Techniques....7
2.4.1 Static Analysis....7
Control Flow....7
Data Flow....7
Information Flow....8
Symbolic Execution....8
Formal Code Verification....8
2.4.2 Formal Analysis....8
2.4.3 Formal Certification....9

3 The Ravenscar Profile Definition....11
3.1 Development History....11
3.2 Definition....11
3.2.1 Ravenscar Features....12
3.3 Summary of Implications of pragma Profile(Ravenscar)....12

4 Rationale....15
4.1 Ravenscar Profile Restrictions....15
4.1.1 Static Existence Model....15
4.1.2 Static Synchronization and Communication Model....16
4.1.3 Deterministic Memory Usage....17
4.1.4 Deterministic Execution Model....17
4.1.5 Implicit Restrictions....19
4.2 Ravenscar Profile Dynamic Semantics....19
4.2.1 Task Dispatching Policy....19
4.2.2 Locking Policy....19
4.2.3 Queuing Policy....19
4.2.4 Additional Run Time Errors Defined by the Ravenscar Profile....19
4.2.5 Potentially-Blocking Operations in Protected Actions....20
4.2.6 Exceptions and the No_Exceptions Restriction....20
4.2.7 Access to Shared Variables....21
4.3 Elaboration Control....22

5 Examples of Use....23
5.1 Cyclic Task....23
5.2 Co-ordinated release of Cyclic Tasks....24
5.3 Cyclic Tasks with Precedence Relations....25
5.4 Event-Triggered Tasks....25
5.5 Shared Resource Control using Protected Objects....26
5.6 Task Synchronization Primitives....27

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

© ISO/IEC 2005 – All rights reserved

5.7 Minimum Separation between Event-Triggered Tasks 28
5.8 Interrupt Handlers 29
5.9 Catering for Entries with Multiple Callers 29
5.10 Catering for Protected Objects with more than one Entry 31
5.11 Programming Timeouts 33
5.12 Further Expansions to the Expressive Power of Ravenscar 34

6 Verification of Ravenscar Programs 37
6.1 Static Analysis of Sequential Code 37
6.2 Static Analysis of Concurrent Code 37
6.2.1 Program-wide Information Flow Analysis 38
6.2.2 Absence of Run-time Errors 39
Elaboration Errors 39
Execution Errors Causing Exceptions 40
Max_Entry_Queue_Length and Suspension Object Check 40
Priority Ceiling Violation Check 40
Potentially Blocking Operations in a Protected Action 41
Task Termination 41
Use of Unprotected Shared Variables 42
6.3 Scheduling Analysis 42
6.3.1 Priority Assignment 42
6.3.2 Rate Monotonic Utilization-based Analysis 43
6.3.3 Response Time Analysis 44
6.3.4 Documentation Requirement on Run-time Overhead Parameters 45
6.4 Formal Analysis of Ravenscar Programs 46

7 Extended Example 47
7.1 A Ravenscar Application Example 47
7.2 Code 50
Cyclic Task 51
Event-response (Sporadic) Tasks 51
Shared Resource Control Protected Object 54
Task Synchronization Primitives 55
Interrupt Handler 57
7.3 Scheduling Analysis 59
7.4 Auxiliary Code 61

8 Definitions, Acronyms, and Abbreviations 67

9 References 73

10 Bibliography 74

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1 Introduction

There is increasing recognition that the software components of critical real-time applications must be provably predictable. This is particularly so for a hard real-time system, in which the failure of a component of the system to meet its timing deadline can result in an unacceptable failure of the whole system. The choice of a suitable design and development method, in conjunction with supporting tools that enable the real-time performance of a system to be analysed and simulated, can lead to a high level of confidence that the final system meets its real-time constraints.

Traditional methods used for the design and development of complex applications, which concentrate primarily on functionality, are increasingly inadequate for hard real-time systems. This is because non-functional requirements such as dependability (e.g. safety and reliability), timeliness, memory usage and dynamic change management are left until too late in the development cycle.

The traditional approach to formal verification and certification of critical real-time systems has been to dispense entirely with separate processes, each with their own independent thread of control, and to use a *cyclic executive* that calls a series of procedures in a fully deterministic manner. Such a system becomes easy to analyse, but is difficult to design for systems of more than moderate complexity, inflexible to change, and not well suited to applications where sporadic activity may occur and where error recovery is important. Moreover, it can lead to poor software engineering if small procedures have to be artificially constructed to fit the cyclic schedule.

The use of Ada has proven to be of great value within high integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. Such subsets have been defined for Ada 83, but these have excluded tasking on the grounds of its non-determinism and inefficiency. Advances in the area of schedulability analysis currently allow hard deadlines to be checked, even in the presence of a run-time system that enforces preemptive task scheduling based on multiple priorities. This valuable research work has been mapped onto a number of new Ada constructs and rules that have been incorporated into the Real-Time Annex of the Ada language standard [RM D]. This has opened the way for these tasking constructs to be used in high integrity subsets whilst retaining the core elements of predictability and reliability.

The Ravenscar Profile is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding only during system integration and testing that the design fails to meet its non-functional requirements.

It is important to note that the Ravenscar Profile is silent on the non-tasking (i.e. sequential) aspects of the language. For example it does not dictate how exceptions should, or should not, be used. For any particular application, it is likely that constraints on the sequential part of the language will be required. These may be due to other forms of *static analysis* to be applied to the code, or to enable *worst-case execution time* information to be derived for the sequential code. The reader is referred to the ISO Technical Report, Guide for the Use of Ada Programming

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Language in High Integrity Systems [GA] for a detailed discussion on all aspects of static analysis of sequential Ada.

The Ravenscar Profile has been designed such that the restricted form of tasking that it defines can be used even for software that needs to be verified to the very highest integrity levels. The Profile has already been included in the ISO technical report [GA] referenced above. The aim of this guide is to give a complete description of the motivations behind the Profile, to show how conformant programs can be analysed and to give examples of usage.

Structure of the Guide

The report is organized as follows. The motivation for the development of the Ravenscar Profile is given in the next chapter. Chapter 3 includes the definition of the profile as agreed by *WG9*; the definition is included here for convenience, but this report is not the definitive statement of the profile. In Chapter 4, the rationale for each aspect of the profile is described. Examples of usage are then provided in Chapter 5. The need for verification is an important design goal for Ravenscar and Chapter 1 reviews the verification approach appropriate to Ravenscar programs. Finally in Chapter 7 an extended example is given. Definitions and references are included at the end of the report.

Readership

This report is aimed at a broad audience, including application programmers, implementers of run-time systems, those responsible for defining company/project guidelines, and academics. Familiarity with the Ada language is assumed.

Conventions

This report uses the *italics* face to flag the first occurrence of terms that have a defining entry in Chapter 8. For all Ada-related terms the report follows the language reference manual [RM] style: it uses the Arial font where there is a reference to defined syntax entities (e.g. delay_relative_statement). For all other names (e.g. Ada.Calendar) it uses normal text font, as do language keywords in the text except that they are in **bold** face.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2 Motivation for the Ravenscar Profile

Before describing the Ravenscar Profile in detail, we will explain in this chapter some of the reasoning behind its features. These primarily come from the need to be able to verify concurrent real-time programs, and to have these programs implemented reliably and efficiently.

In this chapter we look mainly at scheduling theory, as this is the main driver for the definition of the restrictions of the Profile. In addition there is a section that summarizes other program verification techniques that can be used with the Profile.

2.1 Scheduling Theory

Recent research in scheduling theory has found that accurate analysis of real-time behaviour is possible given a careful choice of scheduling/dispatching method together with suitable restrictions on the interactions allowed between tasks. An example of a scheduling method is *preemptive fixed priority scheduling*. Example analysis schemes are *Rate Monotonic Analysis* (*RMA*) [1] and *Response Time Analysis* (*RTA*) [2].

Priority-based preemptive scheduling is usually used with a *Priority Ceiling Protocol* (*PCP*) to avoid unbounded priority inversion and deadlocks. It provides a model suitable for the analysis of concurrent real-time systems. The approach supports *cyclic* and *sporadic* activities, the idea of *hard*, *soft*, *firm*, and *non-critical* components, and controlled inter-process communication and synchronization. It is also scalable to programs for distributed systems.

Tool support exists for RMA and RTA, and for the static simulation of concurrent real-time programs. The primary aim of analysing the real-time behaviour of a system is to determine whether it can be scheduled in such a way that it is guaranteed to meet its timing constraints. Whether the timing constraints are appropriate for meeting the requirements of the application is not an issue for scheduling analysis. Such verification requires a more formal model of the program and the application of techniques such as model checking – see section 2.4.

2.1.1 Tasks Characteristics

The various tasks in an application will each have timing constraints. For *critical tasks* these are normally defined in terms of *deadlines*. The deadline is the maximum time within which a task must complete its operation in response to an event.

Each task is classified into one of the following four basic levels of criticality according to the importance of meeting its deadline:

- Hard
  A *hard deadline task* is one that must meet its deadline. The failure of such a task to meet its deadline may result in an unacceptable failure at the system level.
- Firm
  A *firm deadline task* is one that must meet its deadline under “average” or “normal” conditions. An occasional missed deadline can be tolerated without causing system failure (but may result in degraded system performance). There is no value, and thus there is a system-level degradation of service, in completing a firm task after its deadline.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Soft
  A *soft deadline task* is also one that must meet its deadline under "average" or "normal" conditions. An occasional missed deadline can be tolerated without causing system failure (but may result in degraded system performance). There is value in completing a soft task even if it has missed its deadline.
- Non-critical
  A *non-critical task* has no strict deadline. Such a task is typically a background task that performs activities such as system logging. Failure of a non-critical task does not endanger the performance of the system.

2.1.2 Scheduling Model

At any moment in time, some tasks may be *ready* to run (meaning that they are able to execute instructions if processor time is made available). Others are *suspended* (meaning they cannot execute until some event occurs) or *blocked* (meaning that they await access to a shared resource that is currently exclusively owned by another task). Suspended tasks may become ready synchronously (as a result of an action taken by a currently running task) or asynchronously (as a result of an external event, such as an interrupt or timeout, that is not directly stimulated by the current task).

With priority-based preemptive scheduling on a single processor, a priority is assigned to each task and the scheduler ensures that the highest priority ready task is always executing. If a task with a priority higher than the currently running task becomes ready, the scheduler performs a *context switch*, as soon as it can, to enable the higher-priority task to resume execution. The term "preemptive" indicates that this can occur because of an asynchronous event (i.e. one that is not caused by the running task).

Tasks will normally be required to interact as a result of contention for shared resources, exchange of data, and the need to synchronize their activities. Uncontrolled use of such interactions can lead to a number of problems:

- Unbounded *Priority Inversion /* Blocking
  where a high-priority task is *blocked* awaiting a resource in use by a low-priority task; as a result, ready tasks of intermediate priority may hold up the high priority task for an unbounded amount of time since they will run in preference to the low priority task that has locked the resource.
- *Deadlock*
  where a group of tasks (possibly the whole system) block each other permanently due to circularities in the ownership of and the contention for shared resources.
- *Livelock*
  where several tasks (possibly comprising the whole system) remain ready to run, and do indeed execute, but which fail to make progress due to circular data dependencies between the tasks that can never be broken.
- Missed *Deadline*
  where a task fails to complete its response before its deadline has expired due to factors such as system overload, excessive preemption, excessive blocking, deadlocks, livelocks or CPU overrun.

The restricted scheduling model that is defined by the Ravenscar Profile is designed to minimize the upper bound on blocking time, to prevent deadlocks, and (via tool support) to verify that there is sufficient processing power available to ensure that all critical tasks meet their deadlines.

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
© ISO/IEC 2005 – All rights reserved
Not for Resale

In this model, tasks do not interact directly, but instead interact via shared resources known as *protected objects*. Each protected object typically provides either a resource access control function (including a repository for the private data to manage and implement the resource), or a synchronization function, or a combination of both.

A protected object that is used for resource access control requires a mutual exclusion facility, commonly known as a *monitor* or *critical region*, where at most one task at a time can have access to the object. During the period that a task has access to the object, it must not perform any operation that could result in it becoming suspended. Ada directly supports protected objects and disallows internal suspension within these objects.

A protected object that is used for synchronization provides a signalling facility, whereby tasks can signal and/or wait on events. In the Profile definition, the use of protected objects for synchronization by the critical tasks is constrained so that at most one task can wait on each protected object. A simplified version of wait/signal is also provided in the Profile via the Ada Real-Time Annex functionality known as *suspension objects* [RM D.10]. These can be used in preference to the protected object approach for simple resumption of a suspended task, whereas the protected object approach should be used when more complex resumption semantics are required, for example including deterministic (*race-condition*-free) exchange of data between signaller and waiter tasks.

The Profile definition assures absence of deadlocks by requiring use of an appropriate locking policy. This policy requires a *ceiling priority* to be assigned to each protected object that is no lower than the highest priority of all its calling tasks, and results in the raising of the priority of the task that is using the protected object to this ceiling priority value. In addition to absence of deadlocks, this policy also allows an almost optimal time bound on the worst case blocking time to be computed for use within the schedulability analysis, thereby eliminating the unbounded priority inversion problem. This time bound is calculated as the maximum time that the object is in use by lower-priority tasks. Therefore, the smaller the worst-case time bound for this blocking period, the greater the likelihood that the task set will be schedulable.

The use of priority-based preemptive dispatching defines a mechanism for scheduling. The scheduling policy is defined by the mapping of tasks to priority values. Many different schemes exist for different temporal characteristics of the tasks and other factors such as criticality. What most of these schemes require is an adequate range of distinct priority values. Ada and the Ravenscar Profile ensure this.

2.2 Mapping Ada to the Scheduling Model

The analysis of an Ada application that makes unrestricted use of Ada run-time features including tasking rendezvous, select statements and abort is not currently feasible. In addition, the non-deterministic and potentially unbounded behaviour of many tasking and other run-time calls may make it impossible to provide the upper bounds on execution time that are required for schedulability analysis and simulation. Thus Ada coding style rules and subset restrictions must be followed to ensure that all code within critical tasks is statically time-bounded, and that the execution of the tasks can be defined in terms of response times, deadlines, cycle times, and blocking times due to contention for shared resources.

The application must be decomposed into a number of separate tasks, each with a single thread of control, with all interaction between these tasks identified. Each task has a single primary invocation event. The tasks are categorized as *time-triggered* (meaning that they execute in

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

response to a time event), or *event-triggered* (meaning that they execute in response to a stimulus or event external to the task). If a time-triggered task receives a regular invocation time event with a statically-assigned rate the task is termed *periodic* or *cyclic*.

Protected objects must be introduced to provide mutually-exclusive access to shared resources (e.g. for concurrent access to writable global data) and to implement task synchronization (e.g. via some event signalling mechanism). This decomposition is normally the result of applying a design methodology suitable to describe real-time systems.

In order to be suitable for schedulability analysis, the task set to be analysed must be static in composition and have all its dependencies between tasks via protected objects. Tasks nested inside other Ada structures cause unwanted visibility dependencies and termination dependencies. Therefore, this model only permits tasks to be created at the *library level*, at system initialization time.

This implies that all tasks in the program are created at the library level.

Another consequence of requiring a static task set for schedulability analysis purposes is that the Ravenscar Profile must prohibit the dynamic creation of tasks and protected objects via *allocators*. This implies that the memory requirements for the execution of the task set (e.g. the task stacks) are resolved prior to, or during, elaboration of the program. In addition, the Profile prohibits the implementation from implicitly acquiring dynamic memory from the standard storage pool [RM 13.11(17)]. The data structures that are required by the run-time system should either be declared globally, so that the memory requirements can be determined at link time, or in such a way as to cause the storage to be allocated on the stack (of the *environment task*) during elaboration of the run-time system.

The Profile places no restrictions on the declaration of large or dynamic-sized Ada objects in the application other than prohibiting the implementation from implicitly using the standard storage pool to acquire the storage for these objects. It is acceptable for the memory for such objects to be allocated on the task stack.

2.3 Non-Preemptive Scheduling and Ravenscar

The definition of Ravenscar requires preemptive scheduling of tasks. However a similar profile could be defined that specified non-preemptive execution. Much of the material and guidelines contained in this report would also apply to the non-preemptive case. Non-preemptive implementation for a mono-processor is in between the cyclic executive approach and the preemptive tasking approach with regard to ease of timing analysis, flexibility with regard to change, and responsiveness to asynchronous events. In common with the cyclic executive approach, there is no contention for shared resources, and there is no need to analyse the impact from asynchronous events. There is still, however, the need to break up long code sequences using voluntary suspension points (e.g. a delay_until_statement with a wakeup time argument that denotes a time in the past) to obtain reasonable responsiveness to asynchronous events.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.4 Other Program Verification Techniques

In addition to the provision of support for schedulability analysis, the rationale behind the Ravenscar Profile definition is also to support other static program verification techniques, and to simplify the formal certification process. These other techniques are discussed briefly in this section.

2.4.1 Static Analysis

Static analysis is recognized as a valuable mechanism for verifying software. It is mandated for safety critical applications that are certified to the UK Defence Standard 00-55 [DS]. Industrial experience shows that the use of static analysis during development eliminates classes of errors that can be hard to find during testing. Moreover, these errors can be eliminated by the developer before the code has been compiled or entered into the configuration management system, saving the cost of repeated code review and testing which results from faults that are discovered during *dynamic testing*.

Static analysis as a technology has a fundamental advantage over dynamic testing. If a program property is shown to hold using static analysis, then the property is guaranteed for all scenarios. Testing, on the other hand, may demonstrate the presence of an error, but the correct execution of a test only indicates that the program behaves correctly for the specific set of inputs provided by the test, and within the specific context that the test harness sets up. For all but the simplest systems, exhaustive testing of all possible combinations of input values and program contexts is infeasible. Typically, test cases are devised to represent broad classes of inputs, so that tests can be created that use a representative value from each possible input class. However complex program state contexts are usually only creatable during integration and system testing, when it may be very difficult to simulate all possible operational states. Further, the impact of correcting errors that are found only at this stage of the lifecycle is generally large in comparison to errors found during development.

There are many methods of static analysis. By using combinations of these methods, a variety of properties can be guaranteed for a program. The following list of forms of analysis is drawn from a study of a variety of standards that is presented in the ISO Technical Report [GA]. Section 6.2 discusses how these analyses may be applied in the context of a concurrent Ravenscar Profile program.

Control Flow

Control flow analysis ensures that code is well structured, and does not contain any syntactically or semantically unreachable code.

Data Flow

Data flow analysis ensures that there is no executable path through the program that would result in access to a variable that does not have a defined value. Data flow analysis is only feasible on code that has valid control flow properties.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Information Flow

Information flow analysis is concerned with the dependencies between inputs and outputs within the code. It checks the specified dependencies against the implemented dependencies to ensure consistency. To be effective, information flow analysis needs to be performed with knowledge of the system requirements. It can be a powerful tool for demonstrating properties such as non-interference between critical and non-critical data.

Symbolic Execution

Symbolic execution generates a model of the function of the software in terms of parallel assignments of expressions to outputs for each possible path through the code. This can be used to verify the code without the need for a formal specification.

Formal Code Verification

Formal code verification is the process of proving the code is correct against a formal specification of its requirements. Each operation is specified in terms of the pre-conditions that need to be satisfied for the operation to be callable, and the post-conditions that hold following a successful call to the operation. The verification process demonstrates that, given the pre-conditions, execution of the operation always gives rise to the post-conditions. The level of proof depends on the information provided in the formal specification. This can vary depending on the aspects of the code that need to be verified; this can vary from the proof of a single invariant right up to full functional behaviour.

Proof of absence of run time errors is a special form of formal code verification. This does not require the provision of a formal specification of the program. Instead, formal code verification techniques are used to demonstrate that, at every point in the code where a run-time error may occur, the pre-conditions on execution of that code and the current set of data values in the expression guarantee that the run-time error cannot occur. This is a very valuable property to be able to demonstrate, especially in systems where the occurrence of an unexpected run-time exception is generally unrecoverable, and the overhead of dynamic defensive mechanisms for preventing all such faults is unacceptable.

2.4.2 Formal Analysis

The formal analysis of concurrent programs has been a fruitful research topic for a number of years. Current standard techniques allow many important properties of programs to be statically checked.

Concurrent programs, whilst more expressive than their sequential counterparts, have a number of distinct error conditions that must be addressed during program development. The most common of these is deadlock where all processes are blocked on a synchronization primitive with no processes left to undertake the necessary unblocking actions. In general, a concurrent program should possess two important properties:

1. *Safety* - the system of tasks should not get into an unsafe (undesirable) state (for example; deadlock, livelock).
2. *Liveness* - all desirable states of the task must be reached eventually (that is, useful progress should always be made).

--`,,`-`-`,,`,,`,`,,`---

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

In a real-time concurrent system, ‘liveness’ becomes ‘bounded liveness’ as desirable states must be reached by known deadlines.

Ada, like all other engineering languages, does not have its semantics defined in a formal mathematical way. Hence it is necessary to link a model of the program with the program itself. This link cannot be formal but can be precise. The use of standard patterns for Ada tasks helps this linkage. The formal model could be derived from the code or, more likely in an engineering process, the model is derived from requirements, and the code is obtained via a series of refinements from the model.

There are two general forms for these models and two methods of extracting properties (behaviours) from these descriptions. First, an algebraic form could be used in one of the concurrency languages that does have formally defined semantics; examples being *CSP* and *CCS*. The other, more common approach, is to view the program as a collection of state-transition systems.

Verification comes from either a proof theoretic approach or via model checking. An algebraic description can be proved to be deadlock free, for example, by the use of a theorem prover. Alternatively, a state-transition description (or an algebraic one) can be exercised by an exhaustive search of the set of states the program can enter. This 'checking of the model' can deduce that all safe states, and no unsafe states, can be reached.

The disadvantage of model checking is that an explosion of states can make it impossible to terminate the search. However, there have been considerable (and continuing) advances in the tools for model checking, and now sizeable systems can be verified in a respectably small number of hours of processing time. Theorem proving does not have this problem but it is a more skilled activity and theorem proving tools are not simple to use (i.e. the verification process is not automatic). A proof theoretic approach also has the advantage that it can show that a property is true 'for any number of tasks'; whereas model checking cannot generalize in this way - it will show it is true for six client tasks, say, but for seven the check must be made again. Combinations of proof and model checking are possible and are the subject of current research.

For real-time systems, it is possible to add time to the concurrency model and to then validate temporal aspects of program. Timed versions of formalisms such as CSP exist and state-transition systems with clocks allow timing requirements to be expressed and subsequently verified by model checking. A common formalism for this type of state-transition system is called timed automata. Again, tool support for model checking sets of timed automata is well advanced. One of the very useful features of model checking tools is that they all produce a well-defined counter example for any failed check.

2.4.3 Formal Certification

In order to achieve formal certification of a software architecture and of its Ada implementation, it is necessary to provide verification evidence of safety and reliability of the Ada run-time system as well as for the application-specific components. The run-time system that is needed to implement the dynamic semantics of the full Ada concurrency model is complex, and the number of states that may be represented by its dynamic data structures is large. As a result, it is very challenging for a commercial Ada vendor to produce certification evidence to the highest integrity levels for an entire Ada run-time system.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The Ravenscar Profile definition greatly reduces the size and complexity of the required run-time system so as to simplify the process of providing evidence of its safety and reliability. Ada concurrency features that have major impact on the run-time system semantics, such as abort, asynchronous transfer of control, multiple entry queues each with a list of waiting tasks, requeue statements, task hierarchy and dependency, and *finalization* actions of local protected objects, are eliminated. As a result, it is possible to create not only a small and highly efficient run-time system implementation, but also one that is amenable to the forms of verification applicable to sequential code as described in [GA], which may then be used as evidence to support the formal certification of an entire software system to the highest integrity levels.

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3 The Ravenscar Profile Definition

3.1 Development History

The 8th International Real-Time Ada Workshop (IRTAW) was held in April 1997 at the small Yorkshire village of Ravenscar. Two position papers [3][4] led to an extended discussion on tasking profiles. By the end of the workshop, the Ravenscar Profile had been defined [5] in a form that is almost identical to its current specification.

At the 9th IRTAW [6] (March 1999) the Ravenscar Profile was again discussed at length. The definition was reaffirmed and clarified. The most significant change was the incorporation of Suspension Objects. An Ada Letters paper [5] became the de facto defining statement of the Profile.

By the 10th IRTAW [7] (September 2000) many of the position papers were on aspects of the Profile and its use and implementation. No major changes were made although an attempt to standardize on the Restriction identifiers was undertaken. Time was spent on a non-preemptive version of the Profile. Following the 10th workshop the participants decided to forward the Ravenscar Profile to the ARG – the ISO body in charge of the maintenance of the Ada language – so that its definition could move from a de facto to a real standard. The HRG – the ISO body in charge of the high integrity aspects of the Ada language – was also tasked with producing a Rationale for the Profile, action that has resulted in the production of this guide.

At the 11th IRTAW [8] (April 2002) the formal definition of the profile as formulated by the ARG was agreed. It was confirmed that the Profile requires task dispatching policy FIFO_Within_Priorities and locking policy Ceiling_Locking.

3.2 Definition

The definition of the Profile has now been approved by WG9 for inclusion in the revision of the Ada 95 Standard. The definition is reported here for information only, in a form that matches its latest formal definition by the ARG [AI 249], [AI 305]; in due course, an appropriate WG9 document shall provide the definitive specification for inclusion in the revised language standard.

An application requests the use of the Ravenscar Profile by means of the configuration pragma Profile with the Ravenscar identifier:

```
pragma Profile(Ravenscar);
```

There are, in general, two distinct ways of defining the details of the Profile. Either by defining what is in it, or by declaring those parts of Ada that are not. The ‘official’ definition defines the restrictions that are needed to reduce the full tasking model to Ravenscar. However, this gives a rather negative definition. So we shall first introduce the Profile by focusing on the features it does contain.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3.2.1 Ravenscar Features

Following from the discussion on verification in the previous chapter we are able to define an adequate set of tasking features. The Profile allows programs to contain:

- Task types and objects, defined at the library level.
- Protected types and objects, defined at the library level, with a maximum of one entry per object and with a maximum of one task queued at any time on that entry. The entry barrier must be a single Boolean variable (or a Boolean literal).
- Atomic and Volatile pragmas.
- delay_until_statements.
- Ceiling_Locking policy and FIFO_Within_Priorities dispatching policy.
- The E'Count attribute for protected entries except within entry barriers.
- The Ada.Task_Identification package plus task attributes T'Identity and E'Caller.
- Synchronous task control.
- Task type and protected type discriminants.
- The Ada.Real_Time package.
- Protected procedures as statically bound interrupt handlers.

Together these form a coherent set of features that define an adequate language for expressing the programming needs of statically defined real-time systems.

3.3 Summary of Implications of pragma Profile(Ravenscar)

The following restrictions apply to the alternative mode of operation defined by the Ravenscar Profile. The first set comes from the existing Ada definition of restrictions:

```
Restrictions(Max_Protected_Entries=>1);
Restrictions(Max_Task_Entries=>0);
Restrictions(No_Abort_Statements);
Restrictions(No_Asynchronous_Control);
Restrictions(No_Dynamic_Priorities);
Restrictions(No_Implicit_Heap_Allocations);
Restrictions(No_Task_Allocators);
Restrictions(No_Task_Hierarchy);
```

In addition to these restriction identifiers the dispatching and locking policies defined by the Ravenscar profile are:

```
Task_Dispatching_Policy(FIFO_Within_Priorities);
Locking_Policy(Ceiling_Locking);
```

The following new **pragma** Restrictions identifiers are defined for the Ravenscar Profile.

No_Calendar
: There are no semantic dependencies on package Ada.Calendar.

No_Dynamic_Attachment
: There is no call to any of the operations defined in package Ada.Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, Reference).

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

No_Local_Protected_Objects
Protected objects shall be declared only at library-level.

No_Protected_Type_Allocators
There are no allocators for protected types or types containing protected type components.

No_Relative_Delay
There are no delay_relative_statements.

No_Requeue_Statements
There are no requeue_statements.

No_Select_Statements
There are no select_statements.

No_Task_Attributes_Package
There are no semantic dependencies on package Ada.Task_Attributes.

No_Task_Termination
All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate.

Simple_Barriers
The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.

Max_Entry_Queue_Length
Max_Entry_Queue_Length defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error at the point of the call. For the Ravenscar Profile, the value of this restriction is 1.

Note that the effect of this restriction applies only to protected entry queues due to the accompanying restriction of Max_Task_Entries => 0.

The remainder of the definition concerns errors. The *bounded error* that is the invocation of one of the following potentially blocking operations during a protected action shall be detected:

- a protected entry_call_statement
- a delay_until_statement
- a call to a language-defined subprogram that is potentially blocking, for example Ada.Synchronous_Task_Control.Suspend_Until_True

This is indicated by

```
pragma Detect_Blocking;
```

that forms part of the Ravenscar definition.

Note the detection of these bounded error cases results in Program_Error being raised ([RM] 9.5.1 (17)). Potentially blocking operations that occur in a foreign language domain need not be detected.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The definition of these new restrictions and the motivation for the complete set of restrictions is given in the next chapter. For completeness, the definition of the Ravenscar Profile as it will appear in the amended Ada reference manual is as follows:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);

pragma Locking_Policy(Ceiling_Locking);

pragma Detect_Blocking;

pragma Restrictions(
                  Max_Entry_Queue_Length => 1,
                  Max_Protected_Entries => 1,
                  Max_Task_Entries => 0,
                  No_Abort_Statements,
                  No_Asynchronous_Control,
                  No_Calendar,
                  No_Dynamic_Attachment,
                  No_Dynamic_Priorities,
                  No_Implicit_Heap_Allocations,
                  No_Local_Protected_Objects,
                  No_Protected_Type_Allocators,
                  No_Relative_Delay,
                  No_Requeue_Statements,
                  No_Select_Statements,
                  No_Task_Allocators,
                  No_Task_Attributes_Package,
                  No_Task_Hierarchy,
                  No_Task_Termination,
                  Simple_Barriers);
```


© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4 Rationale

This chapter provides a detailed rationale for the imposition of each restriction and some general discussion about how to work within the restrictions while still retaining flexibility in the design and coding processes.

4.1 Ravenscar Profile Restrictions

4.1.1 Static Existence Model

The restrictions listed below ensure that the set of tasks and interrupts to be analysed is fixed and has static properties (in particular, base priority) after program elaboration. If a variable task set were to exist, then it would be impractical to perform static timing analysis of the program because of the dynamic nature of the requirements for CPU time and the meeting of deadlines.

No_Task_Hierarchy

[RM D.7] *All (nonenvironment) tasks depend directly on the environment task of the partition.*

The restriction No_Task_Hierarchy prevents the declaration of tasks local to procedures or to other tasks. Thus tasks may only be created at the library level, i.e. within the declarative part of library level package specifications and bodies, including child packages and package subunits.

No_Task_Allocators

[RM D.7] *There are no allocators for task types or types containing task subcomponents.*

The restriction No_Task_Allocators prevents the dynamic creation of tasks via the execution of Ada allocators.

No_Task_Termination

[AI 305] *All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate.*

The restriction attempts to mitigate the hazard that may be caused by tasks terminating silently. Real-time tasks normally have an infinite loop as their last statement.

No_Abort_Statements

[RM D.7] *There are no abort_statements, and there are no calls to Task_Identification.Abort_Task.*

The restriction No_Abort_Statements ensures that tasks cannot be aborted. The removal of abort statements (and select then abort) significantly reduces the size and complexity of the run-time system. It also reduces non-determinacy.

No_Dynamic_Attachment

[AI 305] *There is no call to any of the operations defined in package Ada.Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, Reference).*

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The restriction No_Dynamic_Attachment excludes use of the operations in predefined package Ada.Interrupts, which contains primitives to attach and detach handlers dynamically during program execution. In conjunction with restriction No_Local_Protected_Objects (see below) this implies that interrupt handlers can only be attached statically by use of **pragma** Attach_Handler applying to protected procedures within library-level protected objects. Note the types and names defined in Ada.Interrupts can be used.

No_Dynamic_Priorities
[RM D.7] *There are no semantic dependencies on the package Ada.Dynamic_Priorities.*

The restriction No_Dynamic_Priorities disallows the use of the predefined package Ada.Dynamic_Priorities, thereby ensuring that the priority assigned at task creation is unchanged during task execution, except when the task is executing a protected operation and during which time it inherits the ceiling priority.

4.1.2 Static Synchronization and Communication Model

These restrictions are a natural consequence of the static existence model, since a locally declared protected object is meaningless for mutual exclusion and task synchronization purposes if it can only be accessed by one task. Furthermore, a static set of protected objects is required for schedulability analysis.

No_Local_Protected_Objects
[AI 305] *Protected objects shall be declared only at library-level..*

The restriction No_Local_Protected_Objects prevents the declaration of protected objects local to subprograms, tasks, or other protected objects.

No_Protected_Type_Allocators
[AI 305] *There are no allocators for protected types or types containing protected type components.*

The restriction No_Protected_Type_Allocators prevents the dynamic creation of protected objects via Ada allocators.

No_Select_Statements
[AI 305] *There are no select_statements.*

Max_Task_Entries => N
[RM D.7] *Specifies the maximum number of entries per task.*

For the Ravenscar Profile, the value of Max_Task_Entries is zero.

The restrictions Max_Task_Entries => 0 and No_Select_Statements prohibit the use of Ada rendezvous for task synchronization and communication. This ensures that these operations are achieved using only the two supported task synchronization primitives: protected object entries and suspension objects, which both exhibit time-deterministic execution properties needed for static timing analysis.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.1.3 Deterministic Memory Usage

The Profile contains two restrictions that are designed to prevent implicit dynamic memory allocation by the implementation. Note that the Profile does not prevent the use of the standard storage pool or a user-defined storage pool via explicit allocators, although if there were no application-level visibility or control over how the storage in the standard storage pool was managed, the use of this pool would not be recommended.

No_Implicit_Heap_Allocations

[RM D.7] *There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.*

The restriction No_Implicit_Heap_Allocations prevents the implementation from allocating memory from the standard storage pool other than as part of the execution of an Ada allocator.

No_Task_Attributes_Package

[AI 305] *There are no semantic dependencies on the package Ada.Task_Attributes.*

The restriction No_Task_Attributes_Package prevents use of the predefined package Ada.Task_Attributes, which is used to dynamically create attributes of each task in the application. Attribute creation may cause implicit dynamic allocation of memory. Although an implementation is allowed to statically reserve space for such attributes and then to impose a restriction on usage, it is felt that support of this feature is not compatible with the static nature of Ravenscar programs.

4.1.4 Deterministic Execution Model

The following restrictions ensure deterministic execution:

Max_Protected_Entries => N

[RM D.7] *Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.*

For the Ravenscar Profile, the value of Max_Protected_Entries is 1.

Max_Entry_Queue_Length => N

[AI 305] *Specifies the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error exception at the point of the call.*

For the Ravenscar Profile, the value of Max_Entry_Queue_Length is 1, and a call can only be queued on a protected entry, since Max_Task_Entries is 0.

The restrictions Max_Protected_Entries => 1 and Max_Entry_Queue_Length => 1 ensure that at most one task can be suspended waiting on a closed entry barrier for each protected object which is used as a task synchronization primitive. This avoids the possibility of queues of tasks forming on an entry, with the associated non-determinism of the length of the waiting time in the queue. It also avoids two or more barriers becoming open simultaneously as the result of a protected action, with the associated non-determinism of selecting which entry should be serviced first. The restriction also enables a tight time bound on the *epilogue* code to be determined.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The Max_Entry_Queue_Length restriction may only be checkable at run time, in which case violation would result in the raising of the Program_Error exception at the point of the entry call. This is consistent with the Ada rule that states that Program_Error exception is raised upon calling Suspend_Until_True if another task is waiting on that suspension object [RM D.10]. An application could further restrict a Ravenscar program so that only one task is able to call each entry. A static check could then be provided, but this goes beyond what the Profile defines.

Note that, when the restriction Max_Entry_Queue_Length => 1 is in force, **pragma** Queuing_Policy ([RM D.4]) has no effect, since there are no queues.

Simple_Barriers

[AI 305] *The Boolean expression in an entry barrier shall be either a static Boolean expression or a value of a Boolean component of the enclosing protected object.*

The restriction Simple_Barriers, coupled with Max_Protected_Entries => 1, ensures a deterministic execution time and absence of side effects for the evaluation of entry barriers at the epilogue of protected actions within a protected object that is used for task synchronization. There is also scope for additional optimization by the implementation since the barrier value is either static or can be read directly from one of the protected object components, without needing to be computed separately. If the application requires composite entry barrier expressions, this can be achieved by declaring an additional Boolean in the protected data and assigning the composite expression to the Boolean whenever its evaluation result may change. Note the Boolean variable must be declared immediately within the protected object (or type).

No_Requeue_Statements

[AI 305] *There are no requeue_statements.*

The restriction No_Requeue_Statements ensures deterministic task release from protected entry barriers used for task synchronization. The requeue_statement in Ada causes the current caller of a protected entry to be requeued to a different entry dynamically, thereby making it difficult to perform static analysis of task release.

No_Asynchronous_Control

[RM D.7] *There are no semantic dependencies on the package Ada.Asynchronous_Task_Control.*

The restriction No_Asynchronous_Control excludes the use of asynchronous suspension of execution. This ensures that task execution is temporally deterministic. See also the comments made on No_Abort_Statements.

No_Relative_Delay

[AI 305] *There are no delay_relative_statements.*

The restriction No_Relative_Delay prohibits use of the delay_relative_statement based on type Duration. This statement exhibits non-determinism with respect to the absolute time at which the delay expires in the case when the delaying task is preempted after calculating the required relative delay, but before actual suspension occurs. In contrast, the delay_until_statement is deterministic and should be used for accurate release of time-triggered tasks.

No_Calendar

[AI 305] *There are no semantic dependencies on the package Ada.Calendar.*

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The restriction No_Calendar ensures that all timing is performed using the high precision afforded by the time type in package Ada.Real_Time, or by an implementation-defined time type. The Ada.Real_Time time type has a precision of the same order of magnitude as the real-time clock device on the underlying processor board. In contrast, the time type in package Calendar generally has much coarser precision than the real-time clock, due to the need to support a 200 year range, and so its use could result in less accuracy in task release times.

4.1.5 Implicit Restrictions

The set of restriction identifiers for Ada does not represent an orthogonal set of restrictions with the result that some restrictions are implied by others. For example, No_Select_Statements implies Max_Select_Alternatives must be zero.

4.2 Ravenscar Profile Dynamic Semantics

4.2.1 Task Dispatching Policy

The task dispatching policy that is required by **pragma** Profile(Ravenscar) is FIFO_Within_Priorities [RM D.2].

4.2.2 Locking Policy

The locking policy that is required by **pragma** Profile(Ravenscar) is Ceiling_Locking [RM D.3]. This policy provides one of the lowest worst case blocking times for contention for shared resources, and so maximizes the schedulability of the task set when preemptive scheduling is used.

4.2.3 Queuing Policy

The queuing policy is not meaningful for **pragma** Profile(Ravenscar) since no entry queues can form. Thus queuing policy identifiers FIFO_Queuing and Priority_Queuing have no effect.

4.2.4 Additional Run Time Errors Defined by the Ravenscar Profile

The Ada language standard defines a number of concurrency-related run-time checks that may lead to the raising of an exception. The Ravenscar Profile restrictions greatly reduce the quantity of these checks, and thus the number of exception cases that can occur. The two concurrency-related run-time checks that apply to Ravenscar programs are:

- detection of priority ceiling violation as defined by Ceiling_Locking policy;
- detection of violation of not more than one task waiting concurrently on a suspension object (via the Suspend_Until_True operation).

The Ravenscar Profile introduces some additional concurrency-related checks that are potentially detectable only at execution time:

- the maximum number of calls that are queued concurrently on an entry must not exceed one. Program_Error exception is raised if the error occurs (**pragma** Restrictions(Max_Entry_Queue_Length => 1));
- all tasks are non-terminating (**pragma** Restrictions(No_Task_Termination)).

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A conforming implementation must document the effect of a task that attempts to terminate. Possible effects may include:

- allowing the task to terminate silently;
- holding the task in a permanent pre-terminated state;
- executing an application-specific task termination handler via a non-portable feature of the implementation.

Whatever action is taken by the implementation, the application cannot assume that full task termination actions (including finalization) have been executed.

4.2.5 Potentially-Blocking Operations in Protected Actions

The Ravenscar Profile requires detection of the following bounded error in the Ada standard, with the consequential raising of Program_Error exception:

- execution of a potentially-blocking operation during a protected action (**pragma** Detect_Blocking).

The Profile definition does however significantly reduce the list of potentially-blocking operations that may occur during a protected action. In particular, the following potentially-blocking operations are eliminated by the Profile definition:

- a select_statement
- an accept_statement
- a task entry call
- a delay_relative_statement
- an abort_statement
- task creation or activation
- an external requeue_statement with the same target object as that of the protected action.

The Profile definition does not require detection of the potentially blocking operation that is defined by the language standard [RM 9.5.1 (16)]. In this case, it is allowed for the detection to occur at the point of execution of the potentially blocking operation within the called subprogram body.

The rationale for requiring detection of potentially-blocking operations is to allow a highly efficient and temporally deterministic implementation of Ceiling_Locking policy on a mono-processor. In effect, the ceiling priority alone is sufficient to provide the required mutual exclusion without the need to use locks such as *mutexes* once it is guaranteed that the task cannot suspend co-operatively whilst inside the protected operation. This form of locking is also non-queuing on a mono-processor, with the associated benefit of removing the need to compute the worst time that a task may wait in the queue.

4.2.6 Exceptions and the No_Exceptions Restriction

The general concern within high integrity systems of the occurrence of unhandled exceptions is not addressed directly by the Ravenscar Profile since exceptions relate to the sequential, rather than the concurrent, part of the language. Consequently, whereas an unhandled exception will cause a

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

sequential program to terminate, and hence offer an immediate opportunity for some program level control to invoke recovery actions, an unhandled exception during the execution phase of a concurrent program may not be detected. In particular, an unhandled exception can cause any of the following effects:

- silent abandonment of the execution of an interrupt handler;
- silent termination of a task;
- premature exit from a protected action.

The Ravenscar Profile statically avoids the possibility that an exception can be raised by an entry barrier via the restriction Simple_Barriers. In addition, the Profile imposes the restriction No_Task_Termination that requires the implementation to document the effect of a task attempting to terminate. Nevertheless, this is inadequate for most high integrity applications that require static demonstration of absence of exceptions due to run-time check failure. Some techniques are presented in section 6.2 to address the topic of proof of absence of the concurrency-related run-time errors that may occur in a Ravenscar Profile program, using static analysis.

The Ada standard includes the identifier No_Exceptions as a valid argument for the Restrictions pragma. It should be noted that the inclusion of this pragma does not provide a static guarantee of exception freedom - it merely guarantees that the application code does not contain any explicit raise_statement, nor code generation for language-defined checks, nor any exception handlers. However, it is possible for an exception to be raised automatically by the underlying hardware, or by build-in code in the run-time system. There is a documentation requirement on the implementation to define such cases [RM H.4 (25)].

In addition, the language standard defines execution of a program to become *erroneous* if a language-defined check is suppressed via the No_Exceptions restriction and the conditions arise that would have caused the check to fail [RM H.4 (26)]. This is consistent with the suppression of checks using **pragma** Suppress [RM 11.5 (26)]. Since erroneous execution results in the behaviour of a program becoming undefined, the recommendation for high integrity systems is that the No_Exceptions restriction should only be used in conjunction with verification and analysis techniques (see chapter 1) that can statically prove that no exceptions due to run-time check failure can occur. In this case, the No_Exceptions restriction is providing the additional safeguard that exception raising via explicit raise_statements will be prohibited at compile time.

4.2.7 Access to Shared Variables

The Ravenscar Profile requires all synchronization and communication between tasks and interrupt handlers to use data which has mutually-exclusive access. This prevents any erroneous execution that might arise if concurrent access (that includes a write operation) to the same unprotected shared variable is permitted. Such access control is provided in Ada using one of the following constructs:

- a protected object;
- a suspension object;
- an *atomic* object (to which **pragma** Atomic applies).

This access control model applies to the operational phase of the application, after program initialization via elaboration of library-level packages is complete. For each class of object above, it is possible to ensure that its initialization is completed as part of program elaboration.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

There is an issue however in that the semantics of Ada define task activation and interrupt handler attachment to occur during library-level elaboration code for objects that are declared within library-level packages. Consequently it is the case that tasks will execute their declarative part and may proceed into their sequence_of_statements, and that interrupt handlers may execute, prior to the elaboration code for program initialization being completed. This scenario could give rise to the following undesirable effects:

- a task body or interrupt handler may suffer an access-before-elaboration exception;
- a task body or interrupt handler may access uninitialized data;
- a task body or interrupt handler may access unprotected data concurrently that it shares only with the thread of control that is performing the data initialization

It is possible to program each task such that it suspends itself at the start of its sequence of statements, but this is not possible for interrupt handlers (although an application may be able to inhibit interrupts if the device allows). Furthermore, the code executed as part of task activation (prior to the suspension point) may suffer the effects listed above. In order to address this issue, the Partition_Elaboration_Policy pragma has been proposed for the amendment of the Ada standard (see below). If this pragma is used with argument Sequential, then all task activation and interrupt handler attachment is deferred until after all program elaboration code is complete, i.e. just prior to the call of the main subprogram. This pragma complements those that are defined by the Ravenscar Profile to provide the verification that the Profile's goal of controlled access to global shared variables is met during program initialization.

4.3 Elaboration Control

Although not part of Ravenscar, a closely associated new pragma is Partition_Elaboration_Policy [AI 265]. If given the argument Sequential, this defines an alternative elaboration behaviour in which all tasks declared at the library level only proceed to their activation after the environment task has completed all its elaborations and has reached its ‘begin’. All library-based tasks are then activated and executed concurrently. This pragma provides a more deterministic start for a program.

The pragma cannot be used with general Ada programs, but it can be employed with Ravenscar. Note the pragma also prohibits the delivery of interrupts until the environment task has completed its elaboration. This will also be an attractive feature to many users of Ravenscar.

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5 Examples of Use

This chapter illustrates some simple uses of the Ravenscar Profile. The Profile can be used with a variety of coding styles. However if the user is required to perform program analysis, for example to check the schedulability of the tasks, then certain coding styles are recommended. Indeed, a small number of templates can cater for a large class of application needs. In the first eight sections of this chapter we give examples that illustrate the straightforward use of Ravenscar. After that, in sections 5.9 to 5.12, we show how Ravenscar can deal with requirements that would appear to lie outside the scope of what is supported by the Profile.

5.1 Cyclic Task

The task body for a cyclic (or periodic) task typically has, as its last statement, an outermost infinite loop containing one or more delay_until_statements. The basic form of a cyclic task has just a single delay statement either at the start or at the end of the statements within the loop. The Ravenscar Profile supports only one time type for use as the argument – Ada.Real_Time.Time, although a user-defined time type could be used.

Remember that task termination is considered to be an error condition in Ravenscar Profile compliant code since there is no dynamic task creation (and hence the thread of control would be permanently lost). Hence the loop that is presented in the template below is infinite.

A cyclic task will not usually contain any other form of voluntary-suspension statement in the infinite loop, since this would undermine the schedulability analysis

The Ravenscar Profile supports the use of discriminants for task types and protected types. One use of a discriminant is to set differing priorities for task objects or protected objects that are of the same type by using it as the argument of **pragma** Priority. Discriminants can also be used to indicate the period of a cyclic task or other task parameters.

Example 1, Cyclic Template

```
task type Cyclic(Pri : System.Priority; Cycle_Time : Positive) is
  pragma Priority(Pri);
end Cyclic;

task body Cyclic is
   Next_Period : Ada.Real_Time.Time;
   Period : constant Ada.Real_Time.Time_Span :=
                                    Ada.Real_Time.Microseconds(Cycle_Time);
   -- Other declarations
begin
   -- Initialization code
   Next_Period := Ada.Real_Time.Clock + Period;
   loop
      delay until Next_Period; -- wait one whole period before executing
      -- Non-suspending periodic response code
      -- May include calls to protected procedures
      Next_Period := Next_Period + Period;
   end loop;
end Cyclic;

-- now we declare two task objects of this type
C1 : Cyclic(20,200);
C2 : Cyclic(15,100);
```


© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Cyclic tasks normally exchange data through protected operations. In this coding style, there are no protected entries since the only activation event is on **`delay until`**. It is recommended that all shared data be placed in protected objects to avoid corruption.

5.2 Co-ordinated release of Cyclic Tasks

The simple example illustrated above has a number of cyclic tasks that each read the clock and then suspend for time 'period'. It can however by useful for all such tasks to co-ordinate their start times so that they share a common epoch. This can help to enforce precedence relations across tasks. To achieve this a protected object is used which reads the clock on creation and then makes this clock value available to all cyclic tasks.

Example 2, Protected Object Implementing an Epoch

```
protected Epoch is
   function Start_Time return Ada.Real_Time.Time;
private
   pragma Priority(System.Priority'Last);
   Start : Ada.Real_Time.Time := Ada.Real_Time.Clock;
end Epoch;

protected body Epoch is
   function Start_Time return Ada.Real_Time.Time is
   begin
      return Start;
   end Start_Time;
end Epoch;
```

Note, a protected object is not strictly needed as a shared variable appropriately initialized will suffice. A more robust scheme and one that only reads the epoch time once a task actually needs it is as follows.

Example 3, Caller Initialized Epoch

```
protected Epoch is
   procedure Get_Start_Time(T : out Ada.Real_Time.Time);
private
   pragma Priority(System.Priority'Last);
   Start : Ada.Real_Time.Time;
   First : Boolean := True;
end Epoch;

protected body Epoch is
   procedure Get_Start_Time(T : out Ada.Real_Time.Time) is
   begin
      if First then
        First := False;
        Start := Ada.Real_Time.Clock;
      end if;
      T := Start;
   end Get_Start_Time;
end Epoch;
```

This leads to the following further example.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Example 4, Cyclic Task Using Epoch

```
task type Cyclic(Pri : System.Priority; Cycle_Time : Positive) is
  pragma Priority(Pri);
end Cyclic;

task body Cyclic is
   Next_Period : Ada.Real_Time.Time;
   Period : constant Ada.Real_Time.Time_Span :=
                                        Ada.Real_Time.Microseconds(Cycle_Time);
   -- Other declarations
begin
   -- Initialization code
   Epoch.Get_Start_Time(Next_Period);
   Next_Period := Next_Period + Period;
   loop
      delay until Next_Period;   -- wait until next period after epoch
      -- Non-suspending periodic response code
      -- May include calls to protected procedures
      Next_Period := Next_Period + Period;
   end loop;
end Cyclic;
```

5.3 Cyclic Tasks with Precedence Relations

The use of priorities and a shared epoch can be used to enforce precedence, between tasks with the same period, if the application can be restricted so that the tasks do not block during execution. An alternative scheme is to use an offset in time. Here scheduling analysis is used to ensure that each task has completed before the next is released.

Example 5, Cyclic Tasks with Offsets

```
task type Cyclic(Pri : System.Priority; Cycle_Time, Offset : Natural) is
  pragma Priority(Pri);
end Cyclic;

task body Cyclic is
   Next_Period : Ada.Real_Time.Time;
   Period : constant Ada.Real_Time.Time_Span :=
                                  Ada.Real_Time.Microseconds(Cycle_Time);
   -- Other declarations
begin
   -- Initialization code
   Next_Period := Epoch.Start_Time + Ada.Real_Time.Microseconds(Offset);
   loop
      delay until Next_Period;  -- wait until next period after offset
      -- Non-suspending periodic response code
      -- May include calls to protected procedures
      Next_Period := Next_Period + Period;
   end loop;
end Cyclic;

First : Cyclic(20,200,0);  -- required to complete with deadline 70
Second : Cyclic(20,200,70);
```

5.4 Event-Triggered Tasks

The task body for an event-triggered task that conforms to the Profile typically has, as its last statement, an outermost infinite loop containing as the first statement either a call to a protected entry or a call to Ada.Synchronous_Task_Control.Suspend_Until_True using a Suspension Object. The suspension object is used when no other effect is required in the signalling operation; for example, no data is to be transferred from signaller to waiter. In contrast, the protected entry is

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

used for more elaborate event signalling, when additional operations must accompany the resumption of the event-triggered task.

An event-triggered task will not usually contain any other form of voluntary-suspension statement in the infinite loop.

Example 6, An Event-Triggered Task

```
-- A suspension object, SO, is declared in a visible library unit and is
-- set to True in another (releasing) task

task type Sporadic(Pri : System.Priority) is
   pragma Priority(Pri);
end Sporadic;

task body Sporadic is
   -- Declarations
begin
   -- Initialization code
   loop
      Ada.Synchronous_Task_Control.Suspend_Until_True(SO);
      -- Non-suspending sporadic response code
   end loop;
end Sporadic;

Sp : Sporadic(17);
```

5.5 Shared Resource Control using Protected Objects

A protected object used to ensure mutually exclusive access to a shared resource, such as global data, typically contains only protected subprograms as operations, i.e. no protected entries. Protected entries are used only for task synchronization purposes where data exchange is involved. A protected procedure should be used when the internal state of the protected data must be altered, and a protected function should be used for information retrieval from the protected data, when the data remains unchanged.

The Ada Reference Manual states that the use of any form of voluntary-suspension statement during the execution of a protected operation is a bounded error [RM 9.5.1 (8)]. The Ravenscar Profile requires, via **pragma** Detect_Blocking, an implementation to detect this error (and hence to raise Program_Error exception), other than in the case when suspension is due to execution outside of the Ada environment, for example within an underlying operating system call or within imported code that is written in another language.

It is essential to choose the correct value for the ceiling priority of the protected object. By default, the value is System.Priority'Last, unless the protected object contains interrupt handlers (see below). The chosen value must be at least as high as the highest priority task that calls one of the protected operations. If this is not the case, the Ada Reference Manual requires Program_Error exception to be raised when a task with a priority higher than the ceiling priority makes a call to one of the protected operations. However, if the ceiling value is higher than necessary, there may be an increase in the blocking time that high priority tasks will suffer, and consequently a decrease in the overall schedulability of the system. Tool support may be available to determine the optimal ceiling value when the calling sequences can be statically analysed.

--`,,`-`-`,,`,,`,`,,`---

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Example 7, Use of Protected Object for Mutual Exclusion

```
protected Shared_Data is
   function Get return Data;  -- for some global type, Data
   procedure Put(D : in Data);
private
   pragma Priority(10); -- All callers must have priority no greater than 10
   Current : Data;  -- Shared data declaration
end Shared_Data;

protected body Shared_Data is
   function Get return Data is
   begin
      return Current;
   end Get;
   procedure Put(D : in Data) is
   begin
      Current := D;
   end Put;
end Shared_Data;
```

5.6 Task Synchronization Primitives

Task synchronization, in the form of a wait/signal event model, can be achieved in the Ravenscar Profile using either a protected entry or a suspension object, as described above for event-triggered tasks.

The suspension object is the optimized form for a simple suspend/resume operation. The package Ada.Synchronous_Task_Control is used to declare a suspension object, and the primitives Suspend_Until_True and Set_True are used for the suspend and resume operations respectively.

The use of protected objects with entries for task synchronization is restricted by the Ravenscar Profile. The protected object can have at most one entry declaration; the entry barrier must be a simple value that is either a Boolean literal or a Boolean variable that is part of the protected state; and at most one task is allowed to wait on the protected entry at any time. These restrictions provide the necessary determinism in knowing which waiting task is serviced first when barriers become true, since there can be at most one such task. This approach is very similar to the suspension object approach except that:

- Data can be transferred from signaller to waiter atomically (i.e. without risk of a race condition) by use of parameters to the protected operations and additional protected data.
- Additional code can be executed atomically as part of signalling by use of the bodies of the protected operations.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Example 8, Event-Triggered Tasks Suspending on a Protected Entry

```
protected type Event(Ceiling : System.Priority) is
   entry Wait(D : out Data);
   procedure Signal(D : in Data);
private
   pragma Priority(Ceiling);  -- Ceiling priority defined for each object
   Current : Data;  -- Event data declaration
   Signalled : Boolean := False;
end Event;

protected body Event is
   entry Wait(D : out Data) when Signalled is
   begin
      D := Current;
      Signalled := False;
   end Wait;
   procedure Signal(D : in Data) is
   begin
      Current := D;
      Signalled := True;
   end Signal;
end Event;

Event_Object : Event(15);

task Event_Handler is
   pragma Priority(14);  -- i.e. this must be not greater than 15
end Event_Handler;

task body Event_Handler is
   -- Declarations, including D of type Data
begin
   -- Initialization code
   loop
     Event_Object.Wait(D);
     -- Non-suspending event handling code
   end loop;
end Event_Handler;
```

5.7 Minimum Separation between Event-Triggered Tasks

To ensure the timely execution of all tasks in a system it may be necessary to enforce a separation between sporadic tasks so that they cannot execute more frequently than some agreed value. This is easily achieved with a delay_until_statement. Note however that this now introduces a second activation event into the code of the task's outer loop. In general this can make the task more difficult to analyse; but in this example it actually facilitates the analysis by ensuring a minimum separation between task activations.

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Example 9, Event-Triggered Task with Minimum Separation

```
task Event_Handler is
   pragma Priority(14);
end Event_Handler;
task body Event_Handler is
   -- Declarations, including D of type Data
   Minimum_Separation : constant Ada.Real_Time.Time_Span :=
                                        -- some appropriate value
   Next : Ada.Real_Time.Time;
begin
   -- Initialization code
   loop
      Event_Object.Wait(D);
      Next := Ada.Real_Time.Clock + Minimum_Separation;
      -- Non-suspending event handling code
      delay until Next;  -- this ensures minimum temporal separation
   end loop;
end Event_Handler;
```

5.8 Interrupt Handlers

The code of an interrupt handler will often be used to initiate a response in an event-triggered task. This is because the code in the handler itself executes at the hardware interrupt level, and typically the major part of the processing of the response to the interrupt is moved into an event response task, which executes at a software priority level with interrupts fully enabled.

In Example 8 above, if signalling is to be achieved via an interrupt, then procedure Signal is identified as an interrupt handler by **pragma** Attach_Handler. This pragma includes an argument of type Ada.Interrupts.Interrupt_ID that identifies the interrupt to which the handler applies. Note however that procedure Signal must now be defined as a parameterless procedure so as to match the definition of **pragma** Attach_Handler.

The ceiling priority of a protected object that contains an interrupt handler must be in the range of System.Interrupt_Priority.

Example 10, Interrupt Handling via a Protected Entry

```
protected Interrupt_Event is
   entry Wait(D : out Data);
   procedure Signal;  -- Must be parameterless
private
   pragma Attach_Handler(Signal, <interrupt_id> );
   pragma Interrupt_Priority(System.Interrupt_Priority'Last);
      -- Wait and Signal will execute with full interrupt lockout
   Current : Data;  -- Event data declaration
   Signalled : Boolean := False;
end Interrupt_Event;

protected body Interrupt_Event is -- similar to the code in Example 8
   -- except that the setting of Current is obtained via a register during
   -- the execution of Signal rather than as an in parameter
```

5.9 Catering for Entries with Multiple Callers

In this and the following three sections we illustrate how to cater for situations that appear to need more functionality than provided by Ravenscar. In doing this we are not attempting to say that Ravenscar will deal with all situations that full Ada covers. The tasking features of Ada represent a powerful set of abstractions for programming concurrent and real-time systems. To gain

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

predictability and efficiency Ravenscar has had to drop many of these features, and it is not appropriate to try and reintroduce them via a combination of programming tricks and conventions. However there are some situations in which a requirement in just part of a program seems outside of the Profile's definition. These can often be catered for by straightforward techniques that benefit from the other restrictions of Ravenscar.

Here we focus on the requirement for two (or more) tasks to call the same entry of some protected object. As an illustration, consider a situation in which a series of tasks create work items, while others consume them. If more than 10 (say) outstanding items ever accumulate then the two separate event-triggered tasks must be released. An atomicity requirement is that the two tasks are only released if both are available and only when new work items are created.

A non Ravenscar Example

```
protected Controller is
   entry Overload;  -- called by two tasks
   procedure Create;
   procedure Consume;
private
   Work_Items : Integer := 0;
   Released : Boolean := False;
end Controller;

protected body Controller is
   entry Overload when Released is
   begin
   if Overload'Count = 0 then -- barrier is closed when both tasks have left
     Released := False;
   end if;
   end Overload;
   procedure Create is
   begin
      Work_Items := Work_Items + 1;
      Released := (Work_Items > 10 and Overload'Count = 2);
      -- barrier is opened when more than 10 items and both tasks are waiting
   end Create;
   procedure Consume is
   begin
      Work_Items := Work_Items – 1;
   end Consume;
end Controller;
```

In Ravenscar two Controller protected objects are needed, one for each task. To get the required atomicity the second Controller must be called from the first.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Example 11, Using Multiple Protected Objects to Mimic an Entry Queue

```
protected First_Controller is
   entry Overload;  -- called by one task
   procedure Check_Called(OK : out Boolean);
private
   Released : Boolean := False;
end First_Controller;

protected body First_Controller is
   entry Overload when Released is
   begin
      Released := False; -- barrier set to False once task has been released
   end Overload;
   procedure Check_Called(OK : out Boolean) is
   begin
      Released := (Overload'Count = 1);
      OK := Released; -- returns True if task waiting
   end Check_Called;
end First_Controller;

protected Second_Controller is
   entry Overload;  -- called by the other task
   procedure Create;
   procedure Consume;
private
   Work_Items : Integer := 0;
   Released : Boolean := False;
end Second_Controller;

protected body Second_Controller is
   entry Overload when Released is
   begin
      Released := False; -- barrier set to False once task has been released
   end Overload;
   procedure Create is
   begin
      Work_Items := Work_Items + 1;
      if Work_Items > 10 and Overload'Count = 1 then
         First_Controller.Check_Called(Released);
      end if;  -- if Released is true then the first task has been released
               -- and the second one must also be released
   end Create;
   procedure Consume is
   begin
      Work_Items := Work_Items – 1;
   end Consume;
end Second_Controller;
```

Note that once a task calls an entry then, in Ravenscar, it cannot cancel the call hence the above algorithm is safe. In the full language task calls can be cancelled and therefore the above approach would not be guaranteed to work.

5.10 Catering for Protected Objects with more than one Entry

To illustrate the way a two entry protected object can be transformed, consider the standard buffer with one task calling the buffer to extract an item and another task calling it to place items in the buffer. Usually both of these calls must be made via entries in a protected object as the extract call must block if the buffer is empty, and the place call must block if the buffer is full. To comply with the Ravenscar restriction of only one entry in any protected object, a protected object is used for mutual exclusion only and two suspension objects are introduced for the necessary conditional synchronization.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Example 12, A Bounded Buffer Example In Ravenscar

```
package Buffer is
   procedure Place_Item(Item : Some_Type);
   procedure Extract_Item(Item : out Some_Type);
end Buffer;

package body Buffer is
   protected Buff is
      procedure Place(Item    : in Some_Type;
                      Success : out Boolean);
      procedure Extract(Item    : out Some_Type;
                      Success : out Boolean);
   private
      Buffer_Full : Boolean := False;
      Buffer_Empty : Boolean := True;
      -- other declarations
   end Buff;

   Non_Full, Non_Empty : Ada.Synchronous_Task_Control.Suspension_Object;

   procedure Place_Item(Item : Some_Type) is
      OK : Boolean;
   begin
      Buff.Place(Item, OK);
      if not OK then
         Ada.Synchronous_Task_Control.Suspend_Until_True(Non_Full);
         -- note this is a task activation event
         Buff.Place(Item, OK); -- OK must be true
      end if;
      Ada.Synchronous_Task_Control.Set_True(Non_Empty);
   end Place_Item;

   procedure Extract_Item(Item : out Some_Type) is
      OK : Boolean;
   begin
      Buff.Extract(Item, OK);
      if not OK then
         Ada.Synchronous_Task_Control.Suspend_Until_True(Non_Empty);
         -- note this is a task activation event
         Buff.Extract(Item, OK); -- OK must be true
      end if;
      Ada.Synchronous_Task_Control.Set_True(Non_Full);
   end Extract_Item;

   protected body Buff is
      procedure Place(Item    : in Some_Type;
                      Success : out Boolean) is
      begin
         Success := not Buffer_Full;
         if not Buffer_Full then
            -- put Item into Buffer
            Buffer_Empty := False;
            -- set Buffer_Full if appropriate
         end if;
      end Place;
      procedure Extract(Item   : out Some_Type;
                        Success: out Boolean) ) is
      begin
         Success := not Buffer_Empty;
         if not Buffer_Empty then
            -- extract Item from Buffer
            Buffer_Full := False;
            -- set Buffer_Empty if appropriate
         end if;
      end Extract;
   end Buff;
end Buffer;
```


© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.11 Programming Timeouts

There may be situations where a call to a protected object's entry should be retracted after a period of time if the event that should release it has not occurred. In full Ada this would be:

```
select
   PO.Call;
   Timeout := False;
or
   delay until Some_Time;
   Timeout := True;
end select;
```

Identical functionality can be achieved in Ravenscar by the use of an extra task that is event-triggered and a protected object that is used to pass the timeout value to this task. This is illustrated below; note the expansion in code needed to accommodate this effect. The full language clearly has significant superior expressive power in this, and other, areas.

Example 13, Programming Timeouts in Ravenscar

```
protected PO is
   entry Call(Timeout : out Boolean);
   procedure Used_To_Release_Call;
   procedure Too_Late;
private
   Timed_Out : Boolean := False;
   Release : Boolean := False;
end PO;

protected body PO is
   procedure Too_Late is
   begin
      if Call'Count = 1 then
         Timed_Out := True;
         Release := True;
      end if;
   end Too_Late;
   procedure Used_To_Release_Call is
   begin
      Timed_Out := False;
      Release := True;
   end Used_To_Release_Call;
   entry Call(Timeout : out Boolean) when Release is
   begin
      Timeout := Timed_Out;
      Release := False;
      -- further non-suspending code if necessary
   end Call;
end PO;
```

cont…

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Example 13, Programming Timeouts in Ravenscar continued

```
protected Timer_Control is
   entry Wait(Wait_Time : out Ada.Real_Time.Time);
   procedure Set_Time(Wait_Time : Ada.Real_Time.Time);
private
   Timeout : Ada.Real_Time.Time;
   Released : Boolean := False;
end Timer_Control;

protected body Timer_Control is
   entry Wait(Wait_Time : out Ada.Real_Time.Time) when Released is
   begin
      Wait_Time := Timeout;
      Released := False;
   end Wait;
   procedure Set_Time(Wait_Time : Ada.Real_Time.Time) is
   begin
      Timeout := Wait_Time;
      Released := True;
   end Set_Time;
end Timer_Control;

task Timer; -- note this task has more than one activation event

task body Timer is
   T : Ada.Real_Time.Time;
begin
   loop
      Timer_Control.Wait(T);
      delay until T;
      PO.Too_Late;
   end loop;
end Timer;

-- application calls the following
Timer_Control.Set_Time(Some_Time);
PO.Call(Timeout);
```

5.12 Further Expansions to the Expressive Power of Ravenscar

If static timing analysis is not of interest to the application program and a more general model of tasks and interrupts is required, this can still be achieved with reasonable expressive power within the subset definition. However, as noted earlier, Ravenscar is not a substitute for the full language when that level of expressive power is needed.

- Dynamic creation and termination of tasks can be simulated by declaring a pool of event-triggered tasks at program start-up, each containing an infinite loop which has a suspending operation as its first statement, such that its execution can be invoked dynamically by one of the task synchronization primitives. Thus, by changing the settings of suspension objects and entry barriers, it is possible for certain tasks to have their execution disabled whilst others have execution enabled.
- Dynamic exchange of interrupt handlers, often required for applications performing *mode change*, can be simulated by embodying all the different handling code for a particular interrupt in one interrupt handler protected procedure, with each of the different actions being coded as case alternatives in a case statement, dependent on a mode selector. By changing the value of the mode selector, the same handler procedure can perform different response actions at various times during program execution.
- Dynamic task priority change is also generally associated with mode change. This can be simulated by use of a separate event response task for each mode of operation (and

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

assigning a different priority to each task as required), such that the execution of each task that belongs to a dormant mode is suspended until signalled when its mode becomes active.

- A similar effect to requeue can be achieved by completing the protected entry body and returning a status result to the caller, which can then emit a subsequent protected entry call to the intended destination of the requeue statement. If each protected entry is called only by a single task, then this alternative technique does not introduce any race conditions.

Similarly if static timing analysis is not of interest, the classic non-timed rendezvous operations can still be achieved within the subset definition by use of suspension objects for synchronization and protected object entries for communication.

Note that no conditional form of suspension is supported by the subset. This can be simulated if a suspension object is used by polling the state of the suspension object (via the Current_State function in package Ada.Synchronous_Task_Control), or if a protected entry is used by polling the value of the protected data which controls the synchronization (i.e. the barrier Boolean).

--`,,,```-`-`,,`,,`,`,,`---

© ISO/IEC 2005 All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Blank page

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6 Verification of Ravenscar Programs

In chapter 1 the motivation for the Profile was described in terms of the need to verify the temporal behaviour of concurrent real-time programs. In this chapter we give an introduction to the forms of verification that can be used with Ravenscar to deliver dependable systems.

The approach to verification in the presence of Ada tasking is similar in many ways to that traditionally used for cyclic executives. Each thread of control is independently verified for conformance with its precise/formal specification, for example by performing requirements-based testing or by use of static analysis tools on its sequential behaviour. Then, the program as a whole is verified against all its timing constraints. This latter stage differs from the cyclic executive approach in the presence of priority-based preemptive task scheduling in that it can be automated by the use of, for example, a Response Time Analysis (RTA) tool to verify that a given task set meets its deadlines. The tool-based approach greatly simplifies the process of verification of timing constraints during development, and of re-verification after the system has undergone modification during maintenance.

The effects of arbitrary dynamic preemption can be statically analysed by considering all accesses to the global state of the program as being volatile, e.g. two successive reads to the same global state variable may deliver different values (as for reads of values delivered by an external device).

The core set of Ravenscar Profile run-time system packages can be developed to the most stringent software development standards so that these packages are suitable for inclusion in an application that requires certification against an applicable standard such as RTCA DO-178B [DO].

In this chapter we look at four levels of verification:

- Static analysis of sequential code
- Static analysis of concurrent code
- Scheduling analysis
- Formal analysis

6.1 Static Analysis of Sequential Code

As discussed in the introduction, Ravenscar is silent about those features of the sequential language that should be used with the Profile (apart from requiring no implicit use of the heap). Similarly, it is not appropriate here to discuss the forms of static analysis that should be used to verify the functional behaviour of each task. The reader is referred to the ISO Technical Report Guide for the use of Ada in high integrity applications [GA].

6.2 Static Analysis of Concurrent Code

The two main goals of applying static analysis techniques to Ravenscar programs are:

- to obtain the same level of proof and data / information flow analysis for concurrent programs as is currently achievable for a sequential program;
- to obtain proof of absence of the concurrency-related run-time errors, to supplement the proof of absence of run-time errors that is currently achievable for sequential code.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The concurrency-related run-time errors that apply to Ravenscar programs are described in sections 4.2.4 and 4.2.5.

In addition, it is highly desirable if the implementation-defined effect of task termination in the presence of the No_Task_Termination restriction can be eliminated.

The remainder of this section addresses various techniques for producing static analysis evidence to meet the above goals. These verification processes are made possible by the following assertions about the behaviour of a valid Ravenscar program:

- Each task and interrupt handler execution is deferred until after program elaboration is complete.
- Tasks do not terminate.
- All task communication is via protected shared variables (predominantly using protected objects).
- All protected shared variables are initialized during library-level elaboration code.

6.2.1 Program-wide Information Flow Analysis

Current technology supports data flow analysis, information flow analysis, and proof based on pre- and post-conditions and invariants, for sequential code only. The goal is to extend this to Ravenscar programs that include tasks, protected objects and interrupt handlers.

The data dependency information that is currently used to analyse sequential programs can be applied to each task and each interrupt handler in the concurrent program as an independent entity. Thus the existing tools and techniques can verify each thread of control in isolation, including its use of privately accessed global data. This then leaves only the issue of the verification of the interactions between the threads of control as represented by the set of protected shared variables.

The protected shared variables are required to be initialized by the library-level elaboration code in order to ensure that uninitialized shared data is not used. If initialization were instead performed during the operation phase, a race condition could be introduced. For a suspension object, initialization is defined by the Ada standard to occur at the point of declaration. For a protected object or an atomic object, all fields should be initialized either as part of object elaboration, or using library-level package elaboration code. In conjunction with the use of **pragma** Partition_Elaboration_Policy(Sequential) this ensures that no thread of control can access any shared state that has not been fully initialized.

After the initialization phase is complete, the protected shared variables can be modelled for data and information flow analysis purposes if we assume that their data is volatile. Since the data can be updated at any time due to the effects of preemption and interrupt occurrence, any specific task's view of a protected shared variable must assume that the value may change at any time. For example, two successive reads by a task of a protected shared variable may deliver different results and similarly, the value read by a task following a write by the same task cannot be assumed to be the written value. This volatility is the same abstraction as that used to model access to external program data, such as that which has an address clause or is an imported variable (via **pragma** Import). Thus, assuming that the static analysis technique supports access to volatile external data, concurrent access to protected data can be modelled in the same way. As a result, each thread of control can now be described both in terms of its sequential data and information flow, and in terms of its interactions with volatile protected shared variables.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Having obtained the analysis of each thread of control that includes its interactions with the protected state, it is then possible to combine the analyses to form the overall data and information flow for the program as a whole, across the task and interrupt handler boundaries. This allows the designer to make assertions about how the entire program should behave in terms of the effect that it has on its external inputs (including interrupts) to produce its external outputs. These assertions can then be verified by the analysis to the same degree of confidence as is currently achievable in a sequential program.

This form of static analysis does not address the timing or ordering properties of the program. Later sections in this chapter address these topics by describing the use of RTA and other forms of formal analysis, such as model checking, which can prove statically the timing properties of the program.

6.2.2 Absence of Run-time Errors

Existing static analysis techniques can be used to prove absence of run-time errors due to language-defined exceptions within sequential code. The corresponding guidance on the sequential code constructs that may be used to achieve this goal is contained in the Technical Report [GA]. These techniques can be independently applied to each individual thread of control (task, main program or interrupt handler) of a Ravenscar program.

In order to extend these existing techniques to a full Ravenscar program, it is necessary to address the various forms of run-time check failure that relate directly to the concurrency features. These can be broken down into the following groups:

- Errors during program elaboration, such as access-before-elaboration or use of uninitialized data.
- Errors after program elaboration is complete, during the normal operation phase of the application, in particular the exceptions that are cited in sections 4.2.4 and 4.2.5.
- Erroneous behaviour during normal operation, in particular concurrent access to unprotected shared variables (see section 4.2.7).
- Implementation-defined behaviour as a result of violation of the No_Task_Termination restriction.

The following sub-sections discuss various techniques that can be applied to verify statically that these forms of error cannot occur.

Elaboration Errors

Within a sequential program, detection of access before elaboration errors is generally straightforward during program development due to the repeatable nature of the elaboration order, and the raising of Program_Error exception at the point of failure, causing the program to terminate. Having obtained a correct elaboration order during development, this ordering is usually predictable except when a switch to a different compiler vendor, or an upgrade to a new product version from the same vendor that uses a different algorithm for any units that have implementation-defined ordering, is performed. This implicit order variation can be prevented by explicit use of elaboration order pragmas, once a correct order has been established.

Within a concurrent program however, access to global data that is not yet initialized by the elaboration code may occur as a result of race conditions that vary between development mode and deployment mode, due to factors such as the use of hardware of differing performance or memory access times, inclusion or exclusion of checking code, differences in interpretation of priority,

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

scheduling variations etc. These race conditions are more likely to be present because of the Ada rule that a library-level task shall be activated by its master package prior to the execution of that master's body elaboration code, and also prior to the execution of the elaboration code of later library units in the overall program elaboration order. Another contributing factor to the race condition is that having completed its activation, the Ada task proceeds into its normal execution code, and so must be programmed to immediately suspend to prevent this code from executing whilst program elaboration is still incomplete. Similar concerns apply to the execution of interrupt handlers after attachment - an interrupt may trigger execution of a handler prior to completion of program elaboration, and in this case, the handler cannot be programmed to suspend, of course. Such an error may actually occur silently - the task or interrupt handler may read an uninitialized value of a shared variable and not cause any exception to be raised, even in the presence of **pragma** Normalize_Scalars.

There are several solutions that can mitigate this hazard statically. The most obvious one is to ensure that all shared variables of a Ravenscar program are initialized at the point of declaration. However this is inappropriate in the case when elaboration code in the body is needed to set a correct initial value. Logically, it is highly desirable if we can assert that the dynamic semantics of the program are unaffected whether global shared data is initialized at the point of declaration, or by library package body elaboration code, assuming a correct elaboration order for the sequential elaboration code has been enforced using elaboration control pragmas.

In order to achieve the static guarantee that all library units have been elaborated prior to the activation of any task and prior to the invocation of any interrupt handler, the Partition_Elaboration_Policy pragma has been approved for the next revision of the Ada standard. If this pragma is used with argument Sequential, then all task activation and interrupt handler attachment is deferred until after all program elaboration code is complete, i.e. just prior to the call of the main subprogram (see also section 4.2.7).

Execution Errors Causing Exceptions

Sections 4.2.4 and 4.2.5 identify the concurrency-related run-time checks that are required of a conformant implementation of the Profile. In the following sub-sections, we examine techniques for static elimination of these error conditions.

Max_Entry_Queue_Length and Suspension Object Check

The static detection of absence of entry queue length violation may be achieved by applying further constraints on the application code, namely that at most one task object can call each protected entry. This also implies that the task objects, protected objects and protected entries are statically identified. Static identification of an object excludes its name being determined dynamically such as via a function result, a dynamic array index, the dereferencing of an access value etc. A less restrictive scheme that shows that there is no program state in which more than one task may be calling the same protected object would require more extensive analysis, such as the use of model checking (see section 6.4). The same approach can be applied to the static detection of absence of more than one task waiting on each suspension object at any time.

Priority Ceiling Violation Check

The static detection of absence of priority ceiling violation can be achieved assuming the following further constraints:

- all task objects and protected objects have a static priority (this may be supplied via a static expression of a type discriminant for example);

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- the protected object call chain (including nested protected object calls) that is made by each task object and each interrupt handler is statically determinable, by requiring static identification of the target protected object in all cases.

Potentially Blocking Operations in a Protected Action

The static detection of absence of execution of a potentially blocking operation within a protected action is feasible given the additional constraint on the use of indirect subprogram calls, which then allows the call trees to be statically determined. The presence of any of the following constructs in any protected or subprogram body in the call tree that is rooted in a protected operation body would then be statically disallowed:

- a protected entry_call_statement;
- a delay_statement;
- a call to Ada.Synchronous_Task_Control.Suspend_Until_True;
- a call to any other language-defined subprogram that is defined to be potentially blocking [RM 9.5.1 (8-16)].

In addition, the determination of the call trees would enable static detection of an external subprogram call with the same target protected object as that of the protected action, assuming the restriction that the target protected object is always statically identified.

A slightly less restrictive scheme may be possible that uses formal verification methods such as model checking (see section 6.4) to determine if a program state exists such that a protected action would cause execution of a potentially blocking operation (which may be within conditionally-executed code, although this style is not recommended).

It may also be possible to support detection of potentially blocking operations in the presence of indirect procedure calls if a pre-condition that specifies a non-blocking property is asserted prior to each indirect call, and that property is shown to be satisfied statically by all possible procedures that can be invoked by that call. Similarly, the check for circularity in the protected object call chain may be possible even in the case of non-statically identified protected objects, by imposing a pre-condition that none of the potentially called protected objects invoke operations of any protected objects that are higher in the call chain.

Task Termination

The Ravenscar Profile defines a static task set and prohibits dynamic task creation. The intent is that all tasks are created during program start-up, but in any mode of operation, some of them may be dormant, waiting on a synchronization event. A task that is no longer required to be executed would wait on its event indefinitely. In this model, task termination is considered to be an error case and hence the restriction No_Task_Termination is required by the Profile. The effect of violation of the No_Task_Termination restriction is implementation-defined.

Task termination within the restrictions of the Profile can occur only as a result of normal exit from the task body, or as a result of an unhandled exception.

- The case of avoidance of normal exit can be statically analysed if a coding restriction is placed on the task body code - the final statement must either be an infinite loop or else be a compound statement (such as a conditional or case statement) that can only cause an infinite loop to be executed.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The case of showing absence of exceptions by static analysis has already been covered in section 4.2.6 and in the sub-sections above.

The combination of these two techniques can be used to ensure statically that task termination cannot occur, and hence also that no implementation-defined behaviour that results from task termination can be invoked.

Use of Unprotected Shared Variables

The intent of the Profile is that tasks and interrupt handlers should not make concurrent use of an unprotected shared variable - all interactions involving tasks or interrupt handlers are recommended to be via protected and atomic objects, where an atomic object is either a suspension object or one that has **pragma** Atomic applied to it or its type. The avoidance of unprotected shared variables is generally a requirement of high integrity systems, although detection of this erroneous case is not mandated by the Ravenscar Profile definition.

The static detection of absence of unprotected shared variables can be achieved assuming the restriction that the use of all global variables of unprotected type by each task object and by each interrupt handler is statically identifiable. All global objects that are either of a protected type or an atomic type may be safely shared, and so no static identification is required for these. Static verification can then ensure that no unprotected global variable is accessed by more than one thread of control.

Note that if a task object or interrupt handler shares global data only with program elaboration code, i.e. the elaboration code initializes global data that is subsequently privately used by a single task or interrupt handler, then this data does not need to be protected if the Partition_Elaboration_Policy pragma is used with the argument Sequential, since this pragma ensures that the elaboration is complete prior to any task execution or interrupt attachment (and hence there can be no sharing violation).

6.3 Scheduling Analysis

The use of scheduling theory was noted in Chapter 1, here we provide more details on the procedure to be followed. The aim is to introduce the form this analysis takes as it is not appropriate within this report to give a full tutorial on this material; such material can be found in text books (for example [9] and [10]). Ravenscar facilitates the use of these techniques as it supports priority-based dispatching and ceiling locking on protected objects. But, to apply these techniques, further constraints on application code must be made. All tasks must have a single invocation event and allow other parameters to be analysed or measured – see below.

In this section priority assignment is considered first, then two forms of analysis are introduced: Rate Monotonic Analysis and Response Time Analysis.

6.3.1 Priority Assignment

The use of priority-based preemptive dispatching defines a mechanism for scheduling. The scheduling policy is defined by the mapping of tasks to priority values. Many different schemes exist depending on the temporal characteristics of the task and other factors such as criticality. For hard deadline tasks it is usually assumed that the following three parameters are known:

T – Period; time interval between consecutive arrivals of the task

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D – Deadline; required latest completion time for the task (relative to its arrival)

C – Computation time; worst case execution time needed for the task to complete one activation.

For periodic tasks, T is the time interval between releases. For sporadic tasks, T is the minimum inter-arrival time for the event that releases the task. The three parameters (T,D,C) are always given in the same time units. So (30ms, 20ms, 2.73ms) defines a task that (at maximum) is released every 30ms; must complete within 20ms; and that has a maximum computation time of 2.73ms. These latter values are obtained either by measurement or by some form of static timing analysis (or a combination of the two).

If all tasks are hard and criticality itself is not taken into account (because we require all tasks to always meet their deadline) then there is an optimal algorithm for assigning priority if D <= T for all tasks. By optimal we mean that the algorithm is as good as any other fixed priority scheme. The optimal algorithm is called Deadline Monotonic and simply assigns priority based on deadline – the shorter the deadline the higher the priority. In the special case when D = T for all tasks this scheme is known as Rate Monotonic.

An important property of fixed priority dispatching is that the lower priority tasks are the most vulnerable to missing a deadline if there is a run-time problem such as a task executing for more than its assumed maximum C. Because of this property the systems designer may wish to place the highly critical tasks at higher priorities than the Deadline Monotonic scheme would advise. This may reduce schedulability but is perfectly valid and is amenable to Response Time Analysis (see below).

Another reason to raise a task priority is to reduce *jitter* on input and/or output actions. Higher priority tasks have a more regular execution pattern and hence important events such as reading a sensor or writing to an actuator will occur with less variation from one period to the next. Scheduling analysis will only ensure that a task completes somewhere between its release and its deadline. One way of reducing jitter is thus to reduce the deadline of the tasks that perform jitter-sensitive I/O. If this is done then the Deadline Monotonic priority assignment scheme will automatically allocate a higher priority.

Most scheduling schemes assume that each task is assigned a unique priority. Any Ada runtime for Ravenscar will support at least 32 priorities (and may indeed support many more). Although maximum schedulability does require distinct priorities for the tasks, it is unusual for an application to be so close to being unschedulable that it requires these unique priorities. Response Time Analysis can again deal with shared priority values. It should also be noted that that some real-time kernels can exploit the knowledge that tasks share priority to reduce the memory requirement. This is achieved by noting that two (or more) tasks that share a priority level never execute at the same time and hence can ‘share’ a task stack.

Once a priority map has been agreed for the set of tasks within the application the priorities for the protected objects can be assigned systematically.

6.3.2 Rate Monotonic Utilization-based Analysis

For a constrained set of temporal characteristics there exists a very simple schedulability test that quickly verifies if all deadlines will always be met. The constraints are that D=T for all tasks, and that priorities are assigned using the Rate Monotonic scheme. In practice this means that all tasks are hard and periodic. Each task must finish before its next release and there is no additional

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

requirement to control jitter. If we assume, initially, that the program does not contain protected objects (i.e. all tasks execute independently) then the schedulability test is simply a matter of checking the utilization of the task set. For each task the fraction of a complete processor it needs is given by C/T. If this is summed across all tasks this gives the total utilization of the application. Clearly this value must not be more than 1.0 or the system is never going to be schedulable. The actual upper bound (which is less than 1.0) is given by the following formula which is a function of n, the number of tasks in the system.

$$\sum_{i:=1}^{n} \left(\frac{C_i}{T_i} \right) \leq n(2^{1/n} - 1)$$

As n gets arbitrarily large, this expression converges on a single value. This is the famous 'Rate Monotonic' result, which says that a utilization of less than 0.69 will always furnish a schedulable system.

Once protected objects (POs) are introduced, blocking can occur. Here a task when released can be prevented from executing by the currently executing 'low' priority task running with a 'high' ceiling value while in a PO. For each task, the maximum blocking time, B, can be calculated. This is the maximum time a lower priority task can be executing with a priority equal or higher than the task currently under consideration. As noted in Chapter 1, the use of *Immediate Priority Ceiling Protocol* (*IPCP)* on POs does reduce blocking to its minimum value. The utilization test is now augmented with the result that each task must be examined in turn; so for task j:

$$\sum_{i:=1}^{j} \left(\frac{C_i}{T_i} \right) + B_j \leq n(2^{1/j} - 1)$$

Note the blocking term for the lowest priority task is 0 as it cannot suffer blocking.

The simplicity of the utilization-based test makes it a very attractive one to use. But remember, it is for the constrained set of task characteristics. Moreover, it is a necessary but not sufficient test. If the application passes the test all timing constraints will be met. But if it fails the test it may still be schedulable. A better test is needed in these circumstances. The following is one such example.

6.3.3 Response Time Analysis

Response time analysis is a general technique. It will deal with any priority assignment scheme and any relationship between D and T, (although its simple form requires D<=T). Moreover, it is a necessary and sufficient scheme for most situations. Like the utilization-based method it is easily incorporated into tools – many of which already exist.

The form of the analysis is quite straightforward. Firstly, the worst-case (longest) completion time for each task is calculated. This is known as the task response time, R. Secondly, these R values are compared, trivially, with the deadlines to make sure that R is less than D for all tasks. The response time equation is as follows (the *hp* function delivers the set of task with priority higher than task i):

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

--`,,```-`-`,,`,,`,`,,`---

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

As ceiling functions are used, the unit for time is chosen so that all parameters are represented as integers.

The equation is solved by forming a recurrence relation:

$$\omega_i^{k+1} := C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i^k}{T_j} \right\rceil C_j$$

The initial value of the iteration variable is the task's computation time. Iteration continues until either the same value is obtained on two successive iterations (in which case the response time has been calculated) or the value rises higher than the task's deadline (in which case the task is not schedulable).

The above description represents the 'textbook' version of the analysis. The engineering version requires extra terms to capture the *overhead* of actual implementation. Firstly, overheads such as context switches can be assigned to the task that caused them (by incorporating them into the C parameter). Next, the kernel overheads associated with manipulating the delay queue, handling clock interrupts and the releasing of tasks must be factored in. The specific form this takes will depend on the structure of the kernel – but the kernel must provide the data needed to model this overhead. This is a documentation requirement specified in the Real-Time Annex which is discussed further in the following section. For an example on how to include this term in the analysis see the textbooks [9] and [10]. Finally, the overheads incurred by the application's interrupts must be accounted for. We must know a bound on the arrival of such interrupts, and the execution time of each attached handler must be known. Putting these values together allows a set of interrupt overhead terms to be included in the Response Time Analysis.

The appropriate use of the Ravenscar Profile and the scheduling results outlined in the previous three sections provide a sound engineering basis for constructing high integrity real-time systems. The theory is mature and tool support is available.

6.3.4 Documentation Requirement on Run-time Overhead Parameters

There are a number of places in the Reference Manual where documentation requirements and metrics are required of an implementation. Those of most relevance to Ravenscar are:

- C.1 (12 - 20) concerning the interrupt model
- C.3.1 (15,16) concerning overheads of interrupts
- D.2.2 (14 - 16) concerning maximum duration of priority inversion
- D.8 (33 - 45) concerning clock accuracy
- D.9 (8, 11, 13) concerning precision of **delay until**
- D12 (6) concerning interrupt blocking
- D.12 (7) concerning overhead involved with the use of protected objects

Unfortunately, this is not a comprehensive list of the data needed to fully model the overheads caused by the run-time system. Typically also needed are:

- Cost of context switches between tasks
- Cost of handling delay queue operations

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Both of these factors may, depending on the implementation of queues with the run-time system, depend on the number of tasks in the application's program. Nevertheless, if timing analysis is to be used on a Ravenscar program it is necessary to have one of the following:

- Evidence of all necessary parameters
- A means by which the programmer can measure these parameters
- Formulae by which these parameters can be calculated.

6.4 Formal Analysis of Ravenscar Programs

The Ravenscar profile supports only a simple concurrency model with the error conditions being relatively easy to avoid. For example, the use of shared resources (via projected objects with ceiling priorities) cannot lead to deadlock. Nevertheless, to gain a very high level of assurance it may be necessary to formally analyse a Ravenscar program. As outlined in Section 2.4, such analysis takes the form of either mechanized proof (via a theorem prover) or model checking.

There is already experience of using model checking to validate Ravenscar programs. It is possible to add worst-case and best-case execution times for state transitions and to then check that deadlines are never missed. Alternatively, model checking can be used to validate the top-level description of the timing constraints – leaving scheduling analysis to check deadline satisfaction once execution times from the implementation are known. Typical of the verification that can be achieved with this approach is to check some end-to-end deadline through a number of tasks assuming each task itself meets its timing requirements. Each task is represented by an automaton and each protected object by a shared variable (there are no problems with mutual exclusion in these formal models).

As with Ada itself, there can never be a formal map between a Ravenscar program and its model. However, the use of standard paradigms and libraries of associated (reusable) models allows a high integrity process to be defined.

This demonstrates that formal approach can be applied effectively to Ravenscar programs, but this does not imply that all high integrity Ravenscar programs need this level of verification. For many systems, static analysis of each task will be sufficient to generate the appropriate level of confidence.

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7 Extended Example

The example presented in this chapter is designed to illustrate the expressive power of the Ravenscar Profile and the associated coding paradigms that aim to facilitate off-line scheduling analysis.

The example uses all of the concurrency components permitted by the Profile. The structure of the example system models, on a reduced and simplified scale, the operation of real-world embedded real-time systems. The presentation of the example also outlines the information required for, and obtained from, the execution of deadline monotonic priority assignment and off-line scheduling analysis.

7.1 A Ravenscar Application Example

The system in question includes a periodic process that handles orders for a variable amount of workload. Whenever the request level exceeds a certain threshold, the periodic process farms the excess load out to a supporting sporadic process. While such orders are executed, the system may receive interrupt requests from an external source. Each interrupt treatment records an entry in an activation log. When specific conditions hold, the periodic process releases a further sporadic process to perform a check on the interrupt activation entries recorded in the intervening period. The policy of work delegation adopted by the system allows the periodic process to constantly ensure the discharge of a guaranteed level of workload. The correct implementation of this policy also requires that the periodic process is given a higher priority than those assigned to the sporadic processes, so that guaranteed work can be performed in preference to subsidiary activities.

Figure 1, overleaf, shows an HRT-HOOD [11] like representation of the system, while the legend, in figure 1b, recalls the meaning of the symbols and notations used in the diagram.

In HRT-HOOD terms, the system comprises:

- 4 active (i.e. threaded) objects respectively called: Regular_Producer, On_Call_Producer, Activation_Log_Reader, External_Event_Server;
- 1 passive (i.e. unthreaded) object called Production_Workload;
- 3 protected objects respectively called: Request_Buffer, Event_Queue, Activation_Log

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

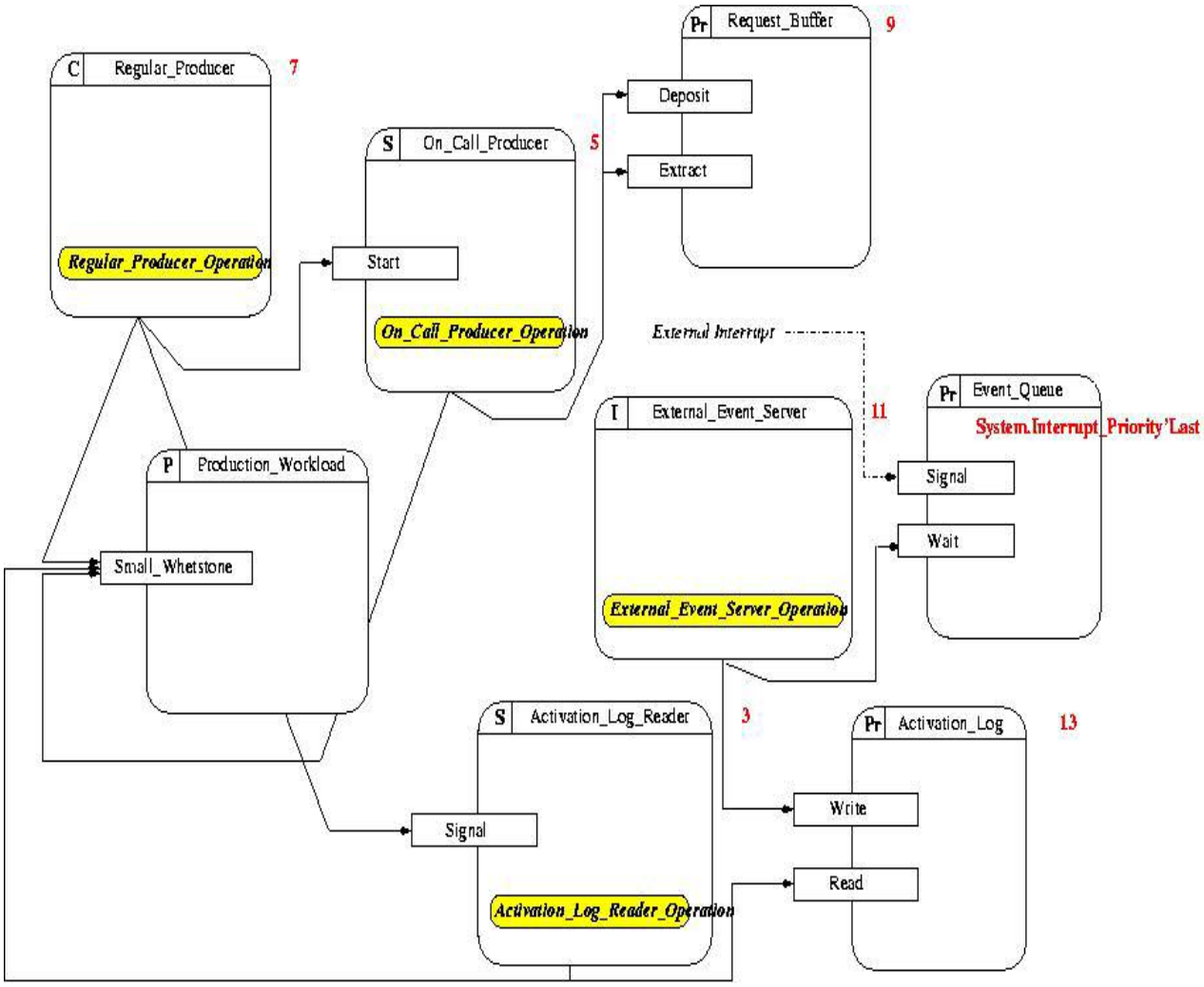

Figure 1: Schematic architecture of the example Ravenscar application.

legend:

Object tags:

Tag	*Position*	*Meaning*
n	Right side of object	Base priority of object internal task or protected object
S	Top–left corner of object	Sporadic object (i.e., threaded object incorporating a sporadic task)
C		Cyclic object (i.e., threaded object incorporating a periodic task)
I		Interrupt sporadic object (i.e., threaded object incorporating an interrupt sporadic task)
Pr		Protected object (i.e., unthreaded object providing protected operations)
P		Passive object (i.e., unthreaded object providing unprotected operations)

Op_Name — Operation exported by the object for users to invoke

Object invocation of exported operation

Op_Name — Internal operation of threaded object

Figure 1b: Legend for the symbols and notations in figure 1.

The operation of the system proceeds as follows:

- Regular_Producer, which figure 1 tags as **C**yclic, embeds a fixed-rate periodic task that carries out a given amount of workload. The example represents the execution of this workload by the invocation of the well-known Small_Whetstone procedure exported by the shared **P**assive object Production_Workload.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- When Regular_Producer determines that the required amount of workload exceeds its ceiling capacity, it delegates the excess workload out to On_Call_Producer. On_Call_Producer, which figure 1 tags as **S**poradic, embeds a sporadic task whose activation is specifically invoked to take over the excess workload of Regular_Producer.
- The sporadic activation and the associated workload transfer occur by means of a typical Ravenscar data-oriented synchronization: Regular_Producer invokes the Start operation exported by On_Call_Producer with a parameter characterising the service request. The Start operation enqueues the request in a private queue embedded within the **Pr**otected object Request_Buffer. We need to protect the buffer because we allow new service requests to come in while the sporadic task is busy executing old ones. This follows from the decision to assign Regular_Producer a higher base priority than that of On_Call_Producer, which we opted for to ensure the discharge of a guaranteed level of workload in preference to the execution of subsidiary activities.
- A successful enqueueing releases the On_Call_Producer sporadic task, which indefinitely waits on an empty queue. The sporadic task fetches the request parameter from the top of the queue and performs the requested amount of workload in the same way as Regular_Producer. An invocation of Start fails when the queue held within Request_Buffer is full; for example, as a result of a (transient) rate of requests faster than service execution. Static analysis of the relationship between the maximum frequency of activation requests and the longest service time incurred by the sporadic task of On_Call_Producer should be used to prevent failure events of this kind.
- While the system carries out the required level of workload (whether regular or excess), an external device may occasionally raise an interrupt to signal its call for attention. In keeping with the Ravenscar programming model, the example application maps the arrival of the external interrupt to the invocation of a protected procedure. Object Event_Queue exports the procedure in question, which we call Signal.
- The service associated with the raising of the interrupt is carried out by the sporadic task embedded in External_Event_Server, which is tagged **I**nterrupt-activated sporadic. To simplify the coding of the example, and in keeping with the programming model that minimizes the amount of activity performed at interrupt priority, we have limited the extent of this interrupt service to the storing of an activation record in a protected buffer. The recording occurs by invocation of procedure Write exported by **Pr**otected object Activation_Log. The use of a protected buffer to hold the activation record offers the natural mechanism to preserve data integrity in the face of independent read and write activities.
- In order for the system to monitor the arrival of service requests from the external device, when certain conditions hold, the periodic process embedded in Regular_Producer requests the task embedded in the **S**poradic object Activation_Log_Reader to examine the latest activation record stored by the interrupt service carried out by External_Event_Server. Activation_Log_Reader does this by invoking the Read procedure of Activation_Log. This style of work partitioning between Regular_Producer and Activation_Log_Reader uses the Ravenscar concurrency mechanisms to allocate activities with differing degrees of importance to distinct tasks. This approach aids system modelling. It also favours the specialization of Ravenscar tasks, which is a way of using the Profile definition to facilitate static analysis of the system.
- The activation request issued by Regular_Producer for this purpose uses the other form of synchronization permitted by the Ravenscar Profile: the data-less synchronization supported by suspension objects. Procedure Signal exported by Activation_Log_Reader performs this synchronization on a suspension object internally held by the object. As HRT-HOOD provides no specific object representation for suspension objects, we have used the

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

convention that procedures by the name Signal exported by **S**poradic objects be understood as implemented by invocation of a private suspension object embedded within the object. Conversely, procedures by the name Start exported by **S**poradic objects are implemented by invocation of the Deposit procedure exported by an associated **Pr**otected object. (Note that Signal is also the name of the protected procedure attached to an interrupt, which dispatches the activation event to **I**nterrupt-activated sporadic objects.)

7.2 Code

The Ravenscar Profile model does not inherently require the application to use any particular coding style for the execution of cyclic and sporadic tasks, protected objects, and interrupt handlers. However, if the application is required to perform schedulability analysis, certain task templates (patterns or stereotypes) and corresponding coding styles are useful in defining the activities that are to be analysed. These task templates were described in Chapter 5 and are used to code the example application outlined above.

Note that, in order to emphasise the stereotype nature of the task templates in the example, we have relegated all the parametric components of the application code into support packages named with "_Parameters" trailer added to the name of the corresponding base package. (The code of these support packages is provided in the closing section of this example.)

The Ravenscar-compliant HRT-HOOD coding convention has individual terminal objects in the system implemented as distinct library-level packages that carry the name of the corresponding object. An HRT-HOOD terminal object is one that cannot be further decomposed and therefore contains at most one type of primitive Ravenscar concurrency component. As each package, associated with a terminal object, by definition contains a single task or protected object, the corresponding entity carries the name of the enclosing package (and thus of the corresponding object).

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Cyclic Task

The example uses one cyclic task, named Regular_Producer, the code of which is shown below. The non-suspending operation of Regular_Producer and its supporting definitions are defined in the Regular_Producer_Parameters package shown at the end of this section.

Regular_Producer

```
with Regular_Producer_Parameters;
package Regular_Producer is
   task Regular_Producer is
      -- assigned by deadline monotonic analysis
      pragma Priority(Regular_Producer_Parameters.Regular_Producer_Priority);
   end Regular_Producer;
end Regular_Producer;

with Regular_Producer_Parameters;
with Ada.Real_Time;
with Activation_Manager;
package body Regular_Producer is
   Period : constant Ada.Real_Time.Time_Span :=
        Ada.Real Time.Milliseconds
        (Regular_Producer_Parameters.Regular_Producer_Period);
   task body Regular_Producer is
      use Ada.Real_Time;
      -- for periodic suspension
      Next_Time : Ada.Real_Time.Time;
   begin
      -- for tasks to achieve simultaneous activation
      Activation_Manager.Synchronize_Activation_Cyclic(Next Time);
      loop
         Next_Time := Next_Time + Period;
         -- non-suspending operation code
         Regular_Producer_Parameters.Regular_Producer_Operation;
         -- time-based activation event
         delay until Next_Time; -- delay statement at end of loop
      end loop;
   exception
      when others =>
         -- last rites: we leave it to "null" for the sake of simplicity
         null;
   end Regular_Producer;
end Regular_Producer;
```

Event-response (Sporadic) Tasks

The example application includes three sporadic tasks, one per type of sporadic activation permitted by the profile: the activation of On_Call_Producer uses a protected object with a suspending entry; the activation of Activation_Log_Reader uses a suspension object; and the activation of External_Event_Server uses a protected object with a suspending entry attached to an interrupt. We first look at the code of the respective sporadic tasks and then turn our attention to the corresponding synchronization objects.

The non-suspending operation of On_Call_Producer and its supporting definitions are defined in the On_Call_Producer_Parameters package shown at the end of this section.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

On_Call_Producer

```
with On_Call_Producer_Parameters;
package On_Call_Producer is
   -- non-suspending operation with queuing of data
   function Start(Activation_Parameter : Positive) return Boolean;
   task On_Call_Producer is
      -- assigned by deadline monotonic analysis
      pragma Priority(On_Call_Producer_Parameters.On_Call_Producer Priority);
   end On_Call_Producer;
end On_Call_Producer;

with Request_Buffer;
with Activation_Manager;
package body On_Call_Producer is
   -- to hide the implementation of the event buffer
   function Start(Activation_Parameter : Positive) return Boolean is
   begin
      return Request_Buffer.Deposit(Activation_Parameter);
   end Start;
   task body On_Call_Producer is
      Current_Workload : Positive;
   begin
      -- for tasks to achieve simultaneous activation
      Activation_Manager.Activation_Sporadic;
      loop
         -- suspending request for activation event with data exchange
         Current_Workload := Request_Buffer.Extract;
         -- non-suspending operation code
         On_Call_Producer_Parameters.On_Call_Producer_Operation
                                                        (Current_Workload);
      end loop;
   exception
      when others =>
         -- last rites
         null;
   end On_Call_Producer;
end On_Call_Producer;
```

The non-suspending operation of Activation_Log_Reader and its supporting definitions are defined in the Activation_Log_Reader_Parameters package shown at the end of this section.

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Activation_Log_Reader

```
with Activation_Log_Reader_Parameters;
package Activation_Log_Reader is
   -- non-suspending parameterless operation
   --+ with no queuing of activation requests
   procedure Signal;
   task Activation_Log_Reader is
      -- assigned by deadline monotonic analysis
      pragma Priority
           (Activation_Log_Reader_Parameters.Activation_Log_Reader_Priority);
    end Activation_Log_Reader;
end Activation_Log_Reader;

with Ada.Synchronous_Task_Control;
with Activation_Manager;
package body Activation_Log_Reader is
   Local_Suspension_Object : Ada.Synchronous_Task_Control.Suspension_Object;
   procedure Signal is
   begin
      Ada.Synchronous_Task_Control.Set_True(Local_Suspension_Object);
   end Signal;
   procedure Wait is
   begin
      Ada.Synchronous_Task_Control.Suspend_Until_True
                                                (Local_Suspension_Object);
   end Wait;
   task body Activation_Log_Reader is
   begin
      -- for tasks to achieve simultaneous activation
      Activation_Manager.Activation_Sporadic;
      loop
         -- suspending parameterless request of activation event
         Wait;
         -- non-suspending operation code
         Activation_Log_Reader_Parameters.Activation_Log_Reader_Operation;
      end loop;
   exception
      when others =>
         -- last rites
         null;
   end Activation_Log_Reader;
end Activation_Log_Reader;
```

The non-suspending operation of External_Event_Server and its supporting definitions are defined in the External_Event_Server_Parameters package shown at the end of this section.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

External_Event_Server

```
with External_Event_Server_Parameters;
package External_Event_Server is
   task External_Event_Server is
      pragma Priority
           (External_Event_Server_Parameters.External_Event_Server_Priority);
   end External_Event_Server;
end External_Event_Server;

with Event_Queue;
with System;
with Activation_Manager;
package body External_Event_Server is
   procedure Wait renames Event_Queue.Handler.Wait;
   task body External_Event_Server is
   begin
      -- for tasks to achieve simultaneous activation
      Activation_Manager.Activation_Sporadic;
      loop
         -- suspending request for external activation event
         Wait;
         -- non-suspending operation code
         External_Event_Server_Parameters.Server_Operation;
      end loop;
   exception
      when others =>
         -- last rites
         null;
   end External_Event_Server;
end External_Event_Server;
```

Shared Resource Control Protected Object

The example application uses one protected object, named Activation_Log, to control access to a shared resource. The auxiliary package Activation_Log_Parameters shown at the end of this section defines all the parameters that characterize the activity of Activation_Log.

--`,,`-`-`,,`,,`,`,,`---

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Activation_Log

```
with Activation_Log_Parameters;
with Ada.Real_Time;
package Activation_Log is
   type Range_Counter is mod 100;
   protected Activation_Log is
      -- must be ceiling of users' priority
      pragma Priority(Activation_Log_Parameters.Activation_Log_Priority);
      -- records interrupt service activation: non-suspending operation
      procedure Write;
      -- retrieves the last activation record: non-suspending operation
      procedure Read
           (Last_Activation  : out Range_Counter;
            Last_Active_Time : out Ada.Real_Time.Time);
   private
      Activation_Counter : Range_Counter := 0;
      Activation_Time : Ada.Real_Time.Time;
   end Activation_Log;
   procedure Write renames Activation_Log.Write;
   procedure Read
        (Last_Activation  : out Range_Counter;
         Last_Active_Time : out Ada.Real_Time.Time)
     renames Activation_Log.Read;
end Activation_Log;

package body Activation_Log is
   protected body Activation_Log is
      procedure Write is
      begin
         Activation_Counter := Activation_Counter + 1;
         Activation_Time := Ada.Real_Time.Clock;
      end Write;
      procedure Read(Last_Activation  : out Range_Counter;
                     Last_Active_Time : out Ada.Real_Time.Time) is
      begin
         Last_Activation := Activation_Counter;
         Last_Active_Time := Activation_Time;
      end Read;
   end Activation_Log;
end Activation_Log;
```

Task Synchronization Primitives

The suspension object is the optimized form for a simple suspend/resume operation. The package Ada.Synchronous_Task_Control is used to declare a suspension object, and the primitives Suspend_Until_True and Set_True are used for the suspend and resume operations respectively. We have seen an example of use of the former in the code of Activation_Log_Reader shown above, whereby the Activation_Log_Reader package exports a Signal procedure that invokes Set_True on the local suspension object on which the Activation_Log_Reader sporadic task suspends by invoking Suspend_Until_True within the call to its internal Wait operation.

As mentioned earlier, the activation of On_Call_Producer is controlled by the use of a protected object named Request_Buffer, which provides a suspending entry named Extract and a releasing procedure named Deposit.

The auxiliary package Request_Buffer_Parameters shown at the end of this section defines all the parameters that characterize the activity of Request_Buffer.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Request_Buffer

```
package Request_Buffer is
   function Deposit(Activation_Parameter : in Positive) return Boolean;
   function Extract return Positive;
end Request_Buffer;

with Request_Buffer_Parameters;
package body Request_Buffer is
   type Request_Buffer_Index is
      mod Request_Buffer_Parameters.Request_Buffer_Range;
   type Request_Buffer_T is array(Request_Buffer_Index) of Positive;
   protected Request_Buffer is
      -- must be ceiling of users' priority
      pragma Priority(Request_Buffer_Parameters.Request_Buffer_Priority);
      procedure Deposit
           (Activation_Parameter : in Positive;
            Response             : out Boolean);
      entry Extract(Activation_Parameter : out Positive);
   private
      My_Request_Buffer : Request_Buffer_T;
      Insert_Index : Request_Buffer_Index := Request_Buffer_Index'First;
      Extract_Index : Request_Buffer_Index := Request_Buffer_Index'First;
      -- the Request_Buffer is initially empty
      Current_Size : Natural := 0;
      -- the guard is initially closed
      -- so that the first call to Extract will block
      Barrier : Boolean := False;
   end Request_Buffer;
   -- we encapsulate the call to protected procedure Deposit in a function
   -- that returns a Boolean value designating the success or failure of
   -- the operation. This coding style allows for a more elegant coding
   -- of the call
   function Deposit(Activation_Parameter : in Positive) return Boolean is
      Response : Boolean;
   begin
      Request_Buffer.Deposit(Activation_Parameter, Response);
      return Response;
   end Deposit;
   -- we encapsulate the call to protected entry Extract in a function
   -- that returns the Positive value designating the workload level passed
   -- by Regular_Producer on to On_Call_Producer. This coding style allows
   -- for a more elegant coding of the call
   function Extract return Positive is
      Activation_Parameter : Positive;
   begin
      Request_Buffer.Extract(Activation_Parameter);
      return Activation_Parameter;
   end Extract;
```

cont…

© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Request_Buffer Continued

```
   protected body Request_Buffer is
      entry Extract(Activation_Parameter : out Positive)
         when Barrier is
      begin
         Activation_Parameter := My_Request_Buffer(Extract_Index);
         Extract_Index := Extract_Index + 1;
         Current_Size := Current_Size - 1;
         -- we close the barrier when the buffer is empty
         -- this also prevents the counter from becoming negative
         Barrier := (Current_Size /= 0);
      end Extract;
      procedure Deposit
           (Activation_Parameter : in Positive;
            Response             : out Boolean) is
      begin
         if Current_Size < Natural(Request_Buffer_Index'Last) then
            My_Request_Buffer(Insert_Index) := Activation_Parameter;
            Insert_Index := Insert_Index + 1;
            Current_Size := Current_Size + 1;
            Barrier := True;
            Response := True;
         else
            -- there is no room for insertion, hence the Deposit returns
            -- with a failure (we might have used as well an over-writing
            -- policy as long as the call returned)
            Response := False;
         end if;
      end Deposit;
   end Request_Buffer;
end Request_Buffer;
```

Interrupt Handler

The example system handles one external interrupt, which is serviced by the interrupt sporadic task External_Event_Server. Event_Queue is the protected object that provides the Signal procedure attached to the interrupt and the Wait suspending entry invoked by External_Event_Server.

The auxiliary package Event_Queue_Parameters shown at the end of the section holds all the definitions required by Event_Queue.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Event_Queue

```
with External_Event_Server_Parameters;
package Event_Queue is
   protected Handler is
      -- must be in the range of System.Interrupt_Priority
      pragma Interrupt_Priority
         (External_Event_Server_Parameters.Event_Queue_Priority);
      procedure Signal;
      entry Wait;
      pragma Attach_Handler
         (Signal,External_Event_Server_Parameters.The_Interrupt);
   private
      -- entry barrier must be simple (i.e. boolean expression)
      Barrier : Boolean := False;
   end Handler;
end Event_Queue;

package body Event_Queue is
   protected body Handler is
      procedure Signal is
      begin
         Barrier := True;
      end Signal;
      entry Wait when Barrier is
      begin
         Barrier := False;
      end Wait;
   end Handler;
end Event_Queue;
```


© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.3 Scheduling Analysis

In order to use the deadline monotonic algorithm to assign priorities to all tasks and protected objects in the above application example we need to determine the respective real-time attributes. This is done in table 1.

Task name	*Task type*	*Period / Minimum interarrival time*	*Deadline*	*Execution time*	*Response time*	*Priority*
Regular_Producer	Cyclic	1000	500			7
On_Call_Producer	Sporadic	≥ 1,000	800			5
Activation_Log_Reader	Sporadic	≥ 1,000	1,000			3
External_Event_Server	Interrupt sporadic	5,000	100			11
Protected object name	***User tasks***				***Ceiling priority***	
Request_Buffer	Regular_Producer (Deposit), On_Call_Producer (Extract)				9	
Event_Queue	External interrupt (Signal), External_Event_Server (Wait)				System.Interrupt_ Priority'Last	
Activation_Log	External_Event_Server (Write), Activation_Log_Reader (Read)				13	

Table 1: Real-time attributes of tasks and protected objects in example application. All time values are in milliseconds.

As soon as we know the worst-case execution time of the non-suspending internal operations performed by the tasks of our example, we can use response time analysis to confirm the feasibility of the real-time attributes of the task set in table 1.

As we mentioned above and as figure 1 illustrates, the example application uses the Small_Whetstone algorithm to control the computational workload of Regular_Producer, On_Call_Producer and Activation_Log_Reader. The way this occurs is shown in the respective auxiliary packages.

Knowing the processing power of the designated target processor and the runtime overheads associated to the execution of the Ravenscar tasking model (e.g. select and context switch time; insert and remove from delay queue; insert and remove from single-position entry queue) we may achieve precise estimates of the required execution time for all tasks and thus allow the use of response time analysis.

By way of example, for one particular assignment of computational workload to the tasks in the system and for the priority assignment shown in table 1, we obtain the schedule of execution shown in figure 2 for the region near the time of system activation (which assumes the arrival of the 1st external interrupt at notional time 0) and in figure 3 for one complete activation of all tasks in the task set.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

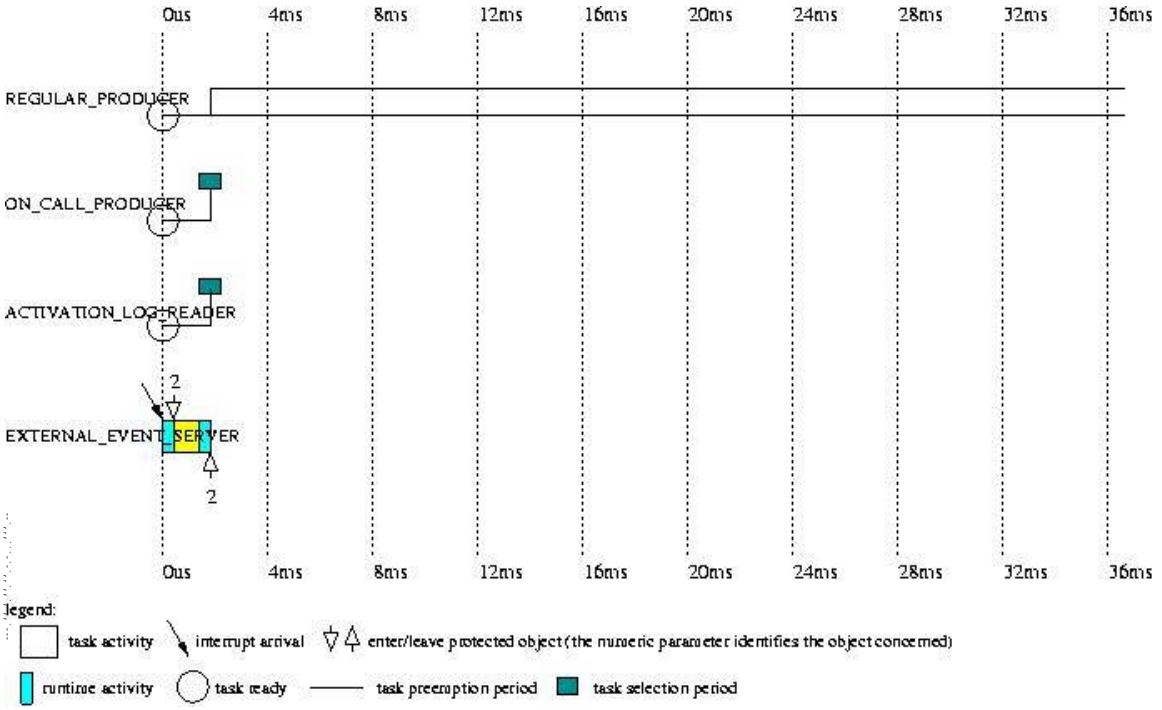

Figure 2: Schedule of task execution near the time of system activation.

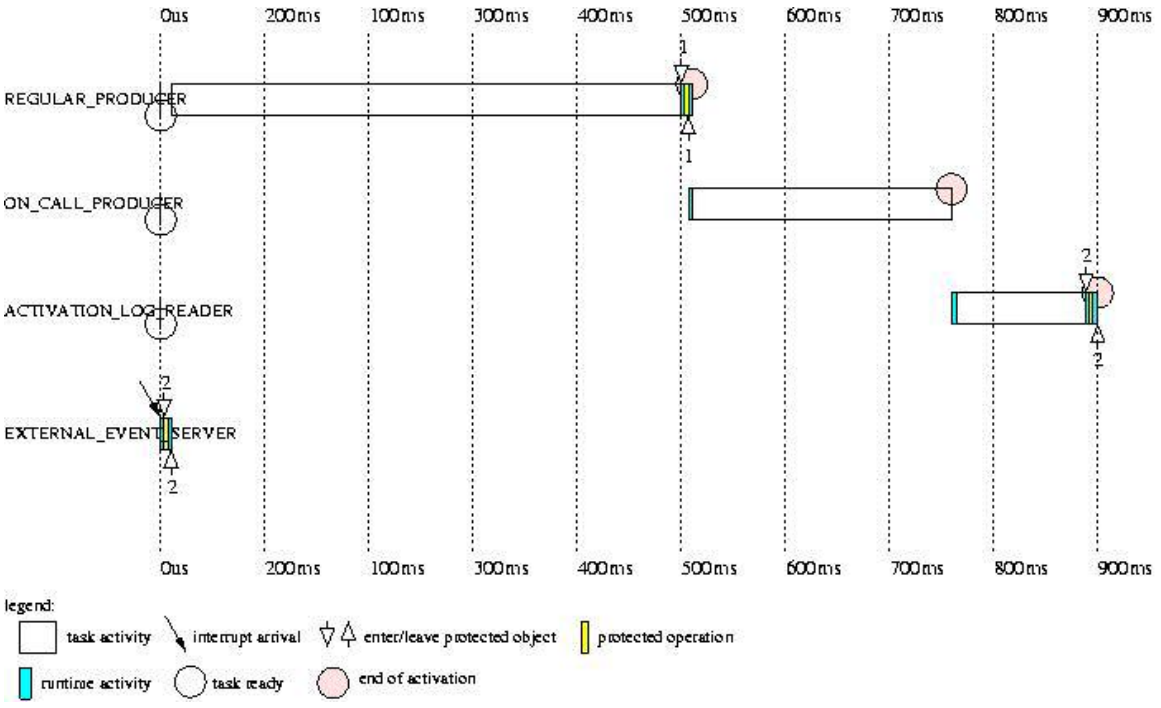

Figure 3: Schedule of execution for one complete activation of all tasks in the example application.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.4 Auxiliary Code

The auxiliary code includes the various operation parameter packages referred to in the earlier descriptions as well as the Activation_Manager.

Regular Producer operation parameters

```
with Auxiliary;
with System;
package Regular_Producer_Parameters is
   Regular_Producer_Priority : constant System.Priority := 7;
   Regular_Producer_Period : constant Natural := 1_000; -- in milliseconds
   procedure Regular_Producer_Operation;
end Regular_Producer_Parameters;

with On_Call_Producer;
with Production_Workload;
with Activation_Log_Reader;
with Ada.Text_IO;
package body Regular_Producer_Parameters is
   -- approximately 5,001,000 processor cycles of Whetstone load
   -- on an ERC32 (a radiation-hardened SPARC for space use) at 10 Hz
   Regular_Producer_Workload : constant Positive := 756;
   -- approximately 2,500,500 processor cycles
   On_Call_Producer_Workload : constant Positive := 278;
   -- the parameter used to query the condition
   -- for the activation of On_Call_Producer
   Activation_Condition : constant Auxiliary.Range_Counter := 2;
   procedure Regular_Producer_Operation is
   begin
      -- we execute the guaranteed level of workload
      Production_Workload.Small_Whetstone(Regular_Producer_Workload);
      -- then we check whether we need to farm excess load out to
      -- On_Call_Producer
      if Auxiliary.Due_Activation(Activation_Condition) then
         -- if yes, then we issue the activation request with a parameter
         -- that determines the workload request
         if not On_Call_Producer.Start(On_Call_Producer_Workload) then
            -- we capture and report failed activation
            Ada.Text IO.Put Line("Failed sporadic activation.");
         end if;
      end if;
      -- we check whether we need to release Activation_Log
      if Auxiliary.Check_Due then
         Activation_Log_Reader.Signal;
      end if;
      -- finally we report nominal completion of the current activation
      Ada.Text_IO.Put_Line("End of cyclic activation.");
   end Regular_Producer_Operation;
end Regular_Producer_Parameters;
```


© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

On_Call_Producer operation parameters

```
with System;
package On_Call_Producer_Parameters is
   On_Call_Producer_Priority : constant System.Priority := 5;
   procedure On_Call_Producer_Operation(Load : Positive);
end On_Call_Producer_Parameters;

with Production_Workload;
with Ada.Text_IO;
package body On_Call_Producer_Parameters is
   procedure On_Call_Producer_Operation(Load : Positive) is
   begin
      -- we execute the required amount of excess workload
      Production_Workload.Small_Whetstone(Load);
      -- then we report nominal completion of current activation
      Ada.Text_IO.Put_Line("End of sporadic activation.");
   end On_Call_Producer_Operation;
end On_Call_Producer_Parameters;
```

Activation_Log_Reader operation parameters

```
with System;
package Activation_Log_Reader_Parameters is
   Activation_Log_Reader_Priority : constant System.Priority := 3;
   procedure Activation_Log_Reader_Operation;
end Activation_Log_Reader_Parameters;

with Production_Workload;
with Activation_Log;
with Ada.Real_Time;
with Ada.Text_IO;
package body Activation_Log_Reader_Parameters is
   -- approximately 1,250,250 processor cycles of Whetstone load
   -- on an ERC32 (a radiation-hardened SPARC for space use) at 10 Hz
   Load : constant Positive := 139;
   procedure Activation_Log_Reader_Operation is
      Interrupt_Arrival_Counter : Activation_Log.Range_Counter := 0;
      Interrupt_Arrival_Time : Ada.Real_Time.Time;
   begin
      -- we perform some work
      Production_Workload.Small_Whetstone(Load);
      -- then we read into the Activation_Log buffer
      Activation_Log.Activation_Log.Read(Interrupt_Arrival_Counter,
         Interrupt_Arrival_Time);
      -- and finally we report nominal completion of current activation
      Ada.Text_IO.Put_Line("End of parameterless sporadic activation.");
   end Activation_Log_Reader_Operation;
end Activation_Log_Reader_Parameters;
```


© ISO/IEC 2005 – All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

External_Event_Server operation parameters

```
with Ada.Interrupts.Names;
with System;
package External_Event_Server_Parameters is
   -- a target-specific interrupt
   The_Interrupt : constant Ada.Interrupts.Interrupt ID :=
        Ada.Interrupts.Names.External_Interrupt_2;
   -- the interrupt priority should be at the appropriate level
   -- (we set it to 'Last because the example handles no other interrupts)
   Event_Queue_Priority : constant System.Interrupt_Priority :=
        System.Interrupt_Priority'Last;
   -- the interrupt sporadic priority is determined by deadline
   -- monotonic analysis
   External_Event_Server_Priority : constant System.Priority := 11;
   procedure Server_Operation;
end External_Event_Server_Parameters;

with Activation_Log;
package body External_Event_Server_Parameters is
   procedure Server_Operation is
   begin
      -- we record an entry in the Activation_Log buffer
      Activation_Log.Write;
   end Server_Operation;
end External_Event_Server_Parameters;
```

Request_Buffer operation parameters

```
with System;
package Request_Buffer_Parameters is
   -- the request buffer priority must ceiling of its users' priorities
   Request_Buffer_Priority : constant System.Priority := 9;
   -- proper analysis will determine the appropriate size of the request
   -- buffer
   Request_Buffer_Range : constant Positive := 5;
end Request_Buffer_Parameters;
```


© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The Activation_Manager provides two facilities.

- A common epoch for all tasks in the system (using the mechanism described in Example 4 of Section 5.2).
- A mechanism for all tasks to suspend until a common time, in order to achieve a co-ordinated release after elaboration. This achieves the effect of **pragma** Partition_Elaboration_Policy(Sequential);.

Activation_Manager internals

```
with Ada.Real_Time;
package Activation_Manager is
   use Ada.Real_Time;
   function Clock return Ada.Real_Time.Time renames Ada.Real_Time.Clock;
   -- global start time relative to which all periodic events
   -- in system will be scheduled
   System_Start_Time : Ada.Real_Time.Time;
   -- relative offset of task activation after elaboration (milliseconds)
   Relative_Offset : constant Natural := 100;
   Task_Start_Time : Ada.Real_Time.Time_Span;
   -- absolute time for synchronization of task activation after elaboration
   Activation_Time : Ada.Real_Time.Time;
   procedure Synchronize_Activation_Sporadic;
   procedure Synchronize_Activation_Cyclic
        (Next_Time : out Ada.Real_Time.Time);
end Activation_Manager;

with System;
package body Activation_Manager is
   procedure Synchronize_Activation_Sporadic is
   begin
      delay until Activation_Time;
   end Synchronize_Activation_Sporadic;
   procedure Synchronize_Activation_Cyclic
        (Next_Time : out Ada.Real_Time.Time) is
   begin
      Next_Time := Activation_Time;
      delay until Activation_Time;
   end Synchronize_Activation_Cyclic;
   procedure Initialize is
      pragma Priority(System.Priority'Last);
   begin
      System_Start_Time := Clock;
      Task_Start_Time := Ada.Real_Time.Milliseconds (Relative_Offset);
      Activation_Time := System_Start_Time + Task_Start_Time;
   end Initialize;
begin
   Initialize;
end Activation_Manager;
```


© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Auxiliary definitions and services

```
package Auxiliary is
   type Range_Counter is mod 5;
   function Due_Activation(Param : Range_Counter) return Boolean;
   type Run_Counter is mod 1_000;
   Factor : constant Natural := 3;
   function Check_Due return Boolean;
end Auxiliary;

package body Auxiliary is
   Request_Counter : Range_Counter := 0;
   Run_Count : Run_Counter := 0;
   -- we establish an arbitrary criterion for the activation of
   -- On_Call_Producer
   function Due_Activation(Param : Range_Counter) return Boolean is
   begin
      Request_Counter := Request_Counter + 1;
      -- we make an activation due according to the caller's input parameter
      return (Request_Counter = Param);
   end Due_Activation;
   -- we establish an arbitrary criterion for the activation of
   -- Activation_Log_Reader
   function Check_Due return Boolean is
      Divisor : Natural;
   begin
      Run_Count := Run_Count + 1;
      Divisor := Natural(Run_Count) / Factor;
      -- we force a check due according to an arbitrary criterion
      return ((Divisor*Factor) = Natural(Run_Count));
   end Check_Due;
end Auxiliary;
```


© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Blank page

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8 Definitions, Acronyms, and Abbreviations

Allocator
An Ada construct used to create an object dynamically [RM 4.8].

Atomic
An operation performed by a task which is guaranteed to produce the same effect as if it were executing in total isolation and without interruption.

Blocked
The state of a task when its execution is prevented, while waiting for mutually-exclusive access to a shared resource which is currently held by a lower priority task.

Bounded error
An implementation- or language-defined error in the application program whose effect is predictable and documented.

Ceiling priority
The priority of a shared resource. The static default priority of all processes that use the resource must be less than or equal to the ceiling priority.

Context switch
The replacement of one task by another as the executing task on a processor.

Critical region
A sequence of statements that must appear to be executed indivisibly.

Critical task
A task whose deadline is significant and whose failure to meet its deadline could cause system failure.

CSP (Communicating Sequential Processes)
A notation for specifying and analyzing concurrent systems.

CSS (Calculus of Communicating Systems)
An algebra for specifying and reasoning about concurrent systems.

Cyclic executive
A scheduler that uses procedure calls to execute each periodic process in a predetermined sequence at a predetermined rate.

Cyclic task
A task whose execution is repeated based on a fixed period of time, also known as a periodic task.

Deadline
The maximum time allowed to a task to produce a response following its invocation.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Deadlock
A situation where a group of tasks (possibly the whole system) block each other permanently.

Dynamic testing
An analysis method that determines properties of the software by observing its execution (cf static analysis).

Erroneous execution
A program state in which execution of the program becomes unpredictable as the result of an error. The errors that result in this state are defined in the language reference manual [RM 1.1.5 (9-10)].

Environment Task
The implicit outermost task which executes the program elaboration code and then calls the main subprogram (if any) [RM 10.2 (8)].

Epilogue
The code executed by the Ada run-time system to service the entry queues as defined in RM 9.5.3(13).

Event-Triggered Task
A task whose invocation is triggered either by an asynchronous action by another task, or by an external stimulus such as an interrupt.

Finalization
An Ada operation which occurs for controlled objects at the point of their destruction [RM 7.6.1].

Firm deadline task
A task whose failure to meet a deadline does not necessarily cause a failure of the application program. There is no value in completing a firm task after its deadline.

Hard deadline task
A task whose failure to meet a deadline may cause a failure of the application program.

IPCP (Immediate Priority Ceiling Protocol, also known as **Priority Ceiling Emulation)**
A technique to minimize the blocking time for contention for shared resources, protected by a protected object. This is provided by the locking policy Ceiling_Locking in Ada [RM D.3].

Jitter
The variation in time between the occurrence of a periodic event and a period of the same frequency.

Library level
The level at which an object which has global accessibility [RM 3.10.2 (22)].

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Livelock
A situation where several tasks (possibly comprising the whole system) remain ready to run, and execute, but fail to make progress.

Liveness
The property that a set of tasks will reach all desirable states.

Mode change
A change in operating characteristics of a system that requires a co-ordinated change in the operation of several different processes in the system.

Monitor
A module containing one or more critical regions; all variables that must be accessed under mutual exclusion are hidden and all procedure calls are guaranteed to execute with mutual exclusion.

Mutex
A locking mechanism used to ensure mutually exclusive access to a shared resource.

Non-critical task
A task with no strict timing requirements.

Overhead
The execution time within the Ada run-time system which must be included in the schedulability analysis.

PBPS (Priority-Based Preemptive Scheduling)
This ensures that, if a high priority task becomes ready to run when a lower priority task is executing on the processor, the high priority task will replace the lower priority task immediately as the executing task.

PCP (Priority Ceiling Protocol)
A set of techniques that bound the blocking time for contention for shared resources. One such protocol, implemented in Ada, is IPCP.

Periodic task
A task whose execution is repeated based on a fixed period of time, also known as a cyclic task.

Preemptive fixed priority scheduling
A scheduling method in which each process has a static priority and the scheduler ensures that the currently selected process is the ready process with the highest priority.

Priority inversion
This occurs when a high-priority task is blocked waiting for a shared resource (including the CPU itself) currently in use by a low-priority task.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Protected object
An Ada construct which is used to provide mutually-exclusive access to shared resources and as a task synchronization primitive.

Race condition
A timing condition that causes processes to operate in an unpredictable sequence so that operation of the system may be incorrect.

Ready
The state of a task when it is no longer suspended. The task, however, will not execute whilst all the available processor resource can be used by higher priority ready tasks.

RMA (Rate Monotonic Analysis)
A mathematical method based on utilization which is used to prove that a set of tasks with static (and simple) characteristics will meet its deadlines in the presence of PBPS.

RTA (Response Time Analysis)
A mathematical method based on calculating latest completion time which is used to prove that a set of tasks with static characteristics will meet its deadlines in the presence of PBPS.

Safety
The property that a set of tasks cannot reach any undesirable state from any desirable state.

Soft deadline task
A task whose failure to meet a deadline does not necessarily cause a failure of the application program. There is value in completing a soft task even if it has missed its deadline.

Sporadic task
An event-triggered task with defined minimum inter-arrival time.

Static analysis
A group of analysis techniques that determine properties of the system from analysis of the program code (c.f. dynamic testing).

Suspended
The state of a task when its execution is stopped due to execution of a language-defined construct that waits for a given time (e.g. a delay statement) or an event.

Suspending operation
An operation which causes the current task to be suspended until released by another task, a timer event or an interrupt handler.

Suspension object
An Ada construct [RM D.10] which is used for basic task synchronization, i.e. suspend and resume, which do not involve data transfer.

Time triggered task
A task whose invocation is triggered by the expiry of a delay set by that task.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

WG9
The Ada Working Group, ISO/IEC JTC1/SC22/WG9. It is the group tasked with the interpretation and maintenance of the Ada Language Standard.

Worst-case execution time
A maximum bound on the time required to execute some sequential code.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Blank page

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9 References

[AI 249] Ravenscar Profile for high integrity systems. ARG, 2002. http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00265.TXT

[AI 265] Partition elaboration policy for high integrity systems. ARG, 2002. http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00265.TXT

[AI 305] New pragma and additional restriction identifiers for real-time systems. ARG, 2002. http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00305.TXT

[DO] DO-178B Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc, Washington D.C. 1992.

[DS] U.K. Ministry of Defence 00-55 Requirements of Safety Related Software in Defence Equipment, 1997.

[GA] Guide for the use of Ada Programming Language in High Integrity Systems, ISO/IEC TR 15942, 2000.

[RM] Ada 95 Reference Manual, International Standard ANSI/ISO/IEC-8652:1995, January 1999.

© ISO/IEC 2005 All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10 Bibliography

[1] Lui, C. and Layland, J., Scheduling algorithms for multiprogramming in a hard real-time environment, JACM, 20 (1), 46 - 61, 1973.

[2] Joseph, M. and Pandya, P., Finding response times in a real-time system, BCS Computer Journal, 29 (5), 390 - 395, 1986.

[3] Burns, A. and Wellings, A.J., Restricted Tasking Models, Ada Letters, XVII (5), 27 - 32, 1997.

[4] Dobbing, B. and Richard-Foy, M., T-SMART - Task Safe, Minimal Ada Realtime Toolset, Ada Letters, XVII (5), 45 - 50, 1997.

[5] Burns, A., The Ravenscar Profile, Ada letters, XIX (4), 49 - 52, 1999.

[6] Session Summary, The Ravenscar Profile and Implementation Issues, Ada Letters, XIX (2), 12 - 14, 1999.

[7] Session Summary, Status and Future of the Ravenscar Profile, Ada Letters, XXI (1), 5 - 8, 2001.

[8] Session Summary, Ravenscar Profile, Proceedings of the 11th International Real-Time Ada Worshop, Ada Letters (to appear in 2003).

[9] Burns, A. and Wellings, A.J., Real-Time Systems and Programming Languages, 3rd Edition, Addison Wesley, 2001.

[10] Liu, J.W.S., Real-Time Systems, Prentice Hall, 2000.

[11] Burns, A. and Wellings, A.J., HRT-HOOD: A design method for hard real-time Ada, Real-Time Systems, 6 (1), 73 - 114, 1994.

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

--`,,```-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

ICS 35.060

Price based on 70 pages

© ISO/IEC 2005 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale