Technical Specification

ISO/IEC TS 24718

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

Technologies de l'information — Langages de programmation — Guide pour l'usage du profil "Ada Ravenscar" dans les systèmes de haute intégrité

First edition
2025-01

Reference number
ISO/IEC TS 24718:2025(en)

Contents

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This first edition cancels and replaces the first edition (ISO/IEC TR 24718:2005).

The main changes are as follows:

— a relatively minor change to the use of a newer syntactic form for specifying aspects of entities, such as the relative priority of a task, rather than the prior use of pragmas;

— a more important change resulting from updates to the definition of the Ravenscar profile, in which support for multiple cores is now included. The primary change is to specify that all assignments of tasks to CPUs are static. In addition, some language-defined facilities are specified as not required or included in the profile for the sake of ensuring a relatively simple run-time library implementation.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

0.1 General

There is an increasing recognition that the software components of critical real-time applications can be demonstrated as predictable. This is particularly the case 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 degradation 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 can occur and where error recovery is important. Moreover, it can lead to poor software engineering if small procedures must be artificially constructed to fit the cyclic schedule.

The use of the Ada programming language 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 ISO/IEC 8652:1987 (conventionally known as "Ada 83" by language users), but these have excluded tasking on the grounds of its non-determinism and inefficiency. Subsequent advances in the area of schedulability analysis have allowed hard deadlines to be checked, even in the presence of a run-time system that enforces pre-emptive 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 (ISO/IEC 8652:2023, Annex D). This evolution has opened the way for these tasking constructs to be used in high integrity subsets while retaining the core elements of predictability and reliability.

The Ravenscar profile is a subset of the tasking model as defined in ISO/IEC 8652:2023. It is restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, and is also suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which can 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. Applicable verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analyses of a system to be performed throughout its development life cycle, thus avoiding the common problem of discovering 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 application in the intended domain, it is likely that constraints on the sequential part of the language will be required. These can 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. See ISO/IEC TR 15942 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 should be verified to the very highest integrity levels. The Ravenscar profile has already been included in ISO/IEC TR 15942.

0.2 Structure

The document is organized as follows. The motivation for the development of the Ravenscar profile is given in Clause 4. Clause 4 also includes the definition of the profile as specified by ISO/IEC 8652:2023 (the Ada

Standard); the definition is included here for convenience, but this document is not the definitive statement of the profile. In Clause 6, the rationale for each aspect of the profile is described. Examples of usage are then provided in Clause 7. The need for verification is an important design goal for the Ravenscar profile: Clause 8 reviews the verification approach appropriate to Ravenscar programs. Finally, in Clause 9 an extended example is given.

0.3 Conventions

For all Ada-related terms, this document follows the style of the ISO/IEC 8652:2023 (the Ada standard): it uses a distinct font where there is a reference to defined syntax entities (e.g. `delay_relative_statement`).

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

1 Scope

This document provides guidance on the use of the Ravenscar profile for concurrent Ada software intended for verification up to, and including, the very highest levels of integrity.

To this end, this document provides a complete description of the motivations behind the Ravenscar profile, to show how conformant programs can be analysed, and to give examples of usage.

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

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1
atomic
type of operation performed by a task that is guaranteed to always produce the same effect as if it were executed in total isolation and without interruption

3.2
blocked
waiting for mutually exclusive access to a shared resource that is currently held by a lower priority task

3.3
bounded error
implementation- or language-defined error in the application program whose effect is predictable and documented

3.4
ceiling priority
static default priority of a shared resource greater than or equal to the highest priority of any accessing task

3.5
context switch
replacement of one task by another as the executing task on a processor

3.6
critical region
sequence of statements that appear to be executed indivisibly

3.7
critical task
task whose *deadline* (3.10) is significant and whose failure to meet its deadline can cause system failure

3.8
cyclic executive
system scheduler that uses procedure calls to execute each periodic process in a predetermined sequence at a predetermined rate

3.9
cyclic task
periodic task
task whose execution is repeated based on a fixed period of time

3.10
deadline
maximum time allowed to a task to produce a response following its invocation

3.11
deadlock
situation in which a group of tasks (possibly the whole system) block each other permanently

3.12
dynamic testing
analysis method that determines properties of the software by observing its execution

Note 1 to entry: See *static analysis* (3.28).

3.13
epilogue
code executed by the Ada run-time system to finish service to protected calls

3.14
event-triggered task
task whose invocation is triggered either by an asynchronous action by another task, or by an external stimulus such as an interrupt

3.15
finalization
operation which occurs for controlled objects at the point of their destruction

Note 1 to entry: See ISO/IEC 8652:2023, 7.6 for further information.

3.16
jitter
variation in time between the occurrence of a periodic event and a period of the same frequency

3.17
livelock
situation in which several tasks (possibly comprising the whole system) remain ready to run, and execute, but fail to make progress

3.18
liveness
system property implying that a set of tasks will reach all desirable states

3.19
mode change
change in operating characteristics of a system that requires a co-ordinated change in the operation of several different processes in the system

3.20
monitor
module containing one or more *critical regions* (3.6) such that all variables potentially accessed under mutual exclusion are hidden and all procedure calls are guaranteed to execute with mutual exclusion

3.21
mutex
locking mechanism used to ensure mutually exclusive access to a shared resource

3.22
overhead
execution time within the Ada run-time system which is included in the schedulability analysis

3.23
priority inversion
situation in which a high-priority task is *blocked* (3.2) waiting for a shared resource (including the processor itself) currently in use by a low-priority task

3.24
race condition
timing condition that causes processes to operate in an unpredictable sequence so that operation of the system can be incorrect

3.25
ready
state of a task when it is no longer *suspended* (3.29) (but not necessarily executing, depending on available processor resources)

3.26
safety
system property implying that a set of tasks cannot reach any undesirable state from any desirable state

3.27
sporadic task
event-triggered task (3.14) with defined minimum inter-arrival time

3.28
static analysis
group of analysis techniques that determine properties of the system from analysis of the program source code

Note 1 to entry: See *dynamic testing* (3.12).

3.29
suspended
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

3.30
suspending operation
operation which causes the current task to be *suspended* (3.29) until released by another task, a timer event or an interrupt handler

3.31
suspension object
Ada construct which is used for basic task synchronization, i.e. suspend and resume, which does not involve data transfer

Note 1 to entry: See ISO/IEC 8652:2023, D.10.

3.32
time-triggered task
task whose invocation is triggered by the expiry of a delay set by that task

3.33
worst-case execution time
maximum bound on the time required to execute some sequential code

4 Motivation for the Ravenscar profile

4.1 General

Before describing the Ravenscar profile in detail, this clause explains 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.

This clause examines mainly scheduling theory, as this is the main driver for the definition of the restrictions of the Ravenscar profile. In addition, there is a subclause that summarizes other program verification techniques that can be used with the profile.

4.2 Scheduling theory

4.2.1 General

State-of-the-art research in scheduling theory has found that accurate analysis of real-time behaviour is possible given a careful choice of scheduling or dispatching method together with suitable restrictions on the interactions allowed between tasks. An example of a scheduling method is fixed priority pre-emptive scheduling, 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. Examples of analysis schemes are rate monotonic analysis (RMA)[5] and response time analysis (RTA).[6]

Fixed-priority pre-emptive scheduling is normally used with a Priority Ceiling Protocol (PCP) to avoid unbounded priority inversion and the risk of circular deadlock on access to shared resources. Adoption of the PCP provides a model suitable for the analysis of concurrent real-time systems. The approach supports cyclic and sporadic activities, the notions of hard, soft, firm, and non-critical application components, and controlled communication and synchronization among application components. It is also scalable to programs for distributed and multiprocessor 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 4.5.

4.2.2 Tasks characteristics

The various tasks in an application will each have timing constraints, which correspond to deadlines.

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 is required to meet its deadline. The failure of such a task to meet its deadline can result in an unacceptable failure at the system level.

— Firm: a firm deadline task is one that is required to meet its deadline under "average" or "normal" conditions. An occasional missed deadline can be tolerated without causing system failure (but can 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.

— Soft: a soft deadline task is also one that is required to meet its deadline under “average” or “normal” conditions. An occasional missed deadline can be tolerated without causing system failure (but can 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.

4.2.3 Scheduling model

At any moment in time, some tasks can be ready to run, meaning that they are able to execute instructions if processor time is made available. Others are suspended, meaning that 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 can 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 pre-emptive scheduling on a mono-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 begin or resume execution. The term “pre-emptive” 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 (also known as 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 can 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 the whole system) remain ready to run, and do indeed execute, but 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 pre-emption, excessive blocking, deadlocks, livelocks or overruns.

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.

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 a maximum of one task at a time can have access to the object. During the period that a task has access to the object, the task is not allowed to perform any operation that can 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 Ravenscar 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 Ravenscar profile via the Ada real-time annex functionality known as suspension objects (see ISO/IEC 8652:2023, 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 Ravenscar profile definition ensures the 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 the 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 pre-emptive 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.

The Ada programming language also provides another facility to help control object sharing: the atomic aspect. All reads and updates applied to an object marked with the atomic aspect are indivisible. Moreover, all tasks of the program (on all processors) that read or update an object marked atomic will see the same order of updates, as that marking also makes the object volatile. The language defines such reads and updates as interactions of the program with the external environment (memory in this case). However, for safe sharing of atomic objects, static assurance of a proper read/write protocol is highly recommended. In order to map Ada to the scheduling model being discussed here, however, protected objects are the primary and preferable abstraction as they are inherently safe.

4.3 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 can make it impossible to provide the upper bounds on execution time that are used to perform schedulability analysis and scheduling simulation. Thus, Ada coding style rules and subset restrictions should 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 is 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 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 are 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 should be static in composition with all its dependencies between tasks set via protected objects. Tasks nested inside other Ada structures incur unwanted visibility dependencies and termination dependencies. Therefore, this model only permits tasks to be created at the library level, at system initialization time.

Hence, in the Ravenscar profile, all tasks in the program are created at the library level.