Safety Critical Software Using Ada
# Table of Contents

**Introduction** .................................................. 1  
- Example of Failure ........................................ 2  
- What is Safety? ............................................. 2  
- Criticality Levels ......................................... 3  
- Standards ...................................................... 5  
  - Avionics – RTCA / EUROCAE ......................... 5  
  - ISO 9000 ..................................................... 8  
  - Def-Stan 0055 ............................................. 8  
  - Development Guidelines for Vehicle-Based Software . 9  
  - IEC 880 ...................................................... 10  

**Use of Ada in Safety Critical Systems** ......................... 12  

**Restricting Ada for Safety Critical Systems** ................ 17  
- Tasking ......................................................... 17  
- Exception Declarations .................................. 18  
- Other Restrictions ......................................... 18  
- Run-Time Requirements .................................. 19  

**Ravenscar Profile** ............................................. 20  
- Cyclic Executives ......................................... 21  
- Periodic Tasks ............................................. 24  
- Event Driven Tasks ....................................... 26  
- Benefits of the Ravenscar Profile .................... 27  

**Ada 95 for Safety Critical Systems** .......................... 29  
- Annex H - Safety and Security .......................... 29  
- HRG .............................................................. 30  

**Certification** .................................................. 32  
- What is Required for Certification ...................... 32  
- Planning Documents ....................................... 32  
  - Plan for Software Aspects of Certification (PSAC) .... 32  
  - Software Development Plan (SDP) ...................... 33  
  - Software Verification Plan (SVP) ...................... 35  
  - Software Configuration Management Plan (SCMP) .... 35  
  - Software Quality Assurance Plan (SQAP) ............ 36  
- Standards ...................................................... 36
Introduction

Software now pervades almost every aspect of daily life. Transport systems depend on software for control of vehicles and their infrastructure. Financial institutions rely on software for accounting and the transfer of money. Industrial software controls equipment and manufacturing processes. Hospitals depend on software for managing patient records and for control of life-support systems. The use of software has grown dramatically over the last decade with the availability of low-cost, high-performance hardware. It is clear that the safety of much human life and property depends directly or indirectly upon the correctness and deterministic properties of software.

Software can provide users with considerable operational flexibility. However, this flexibility brings with it a greatly increased chance of error. There is now an increasing awareness that strict control is needed in order to reduce the risks of errors in what has come to be called safety critical software – that is, software systems whose failure may lead to loss of life or severe injury.

As a result, there is a growing concern in all major industrial nations regarding the legal and ethical obligations of companies and their officers to ensure that systems do not violate safety regulations.

Many industries are in the process of setting, or have already set, specific standards for the development, testing, and certification of safety critical software. As these standards emerge, the focus is on the use of best practice. In some areas, standards mandate specific techniques for the development of safety critical systems. In all cases, a reasoned justification for the techniques actually used is required, together
with evidence to show that the life cycle development processes are being followed.

**Example of Failure**

A passenger airplane is circling in a prearranged location off the coast of Florida. The landing is delayed because of bad weather conditions. As the plane is banking into a turn, a sudden updraft causes the plane to roll much faster than the software control system expects. The software “assumes” a glitch, and the computers are set into an automatic reboot process. The pilot looks on with horror as all of the navigation displays turn blue with a white line through them. At a most crucial moment, when the pilot needs information to stabilize the aircraft, the computers are performing memory checks and restarting the display software.

Fortunately, the pilot has enough height and time to fly the plane by “feel” until the displays are functioning correctly. Had this error occurred when the plane was landing, the consequences could have been catastrophic.

**What is Safety?**

MIL-STD 882B (1984) defines *safety* as follows:

> Freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property.

The terms *safety, reliability, security,* and *correctness* should not be confused. Leveson [Leveson 86] defines the differences among these terms as follows:

> In general, reliability requirements are concerned with making a system *failure free,* whereas safety requirements are concerned with making it *mishap free.*
Reliability is concerned with every possible software error, whereas safety is only concerned with those that result in actual system hazards . . .

Even if all failures cannot be prevented, it may be possible to ensure that the failures that do occur are minor consequences, or that even if a potentially serious failure does occur, the system will, "fail safe" . . .

Unfortunately, in many complex systems, safety and reliability may imply conflicting requirements, and thus a system cannot be built to maximize both . . .

Software is a set of instructions and data that makes a general-purpose computer into an application specific one. Software itself is neither safe nor unsafe. However, if software controls the functionality of a safety critical system, then it becomes safety critical software.

A study of system safety is described in “Safeware - System Safety and Computers” by Nancy Leveson [Leveson 95]. An extensive discussion of safety critical systems, which are usually also real-time control systems, is found in the reference [Pyle 91].

**Criticality Levels**

Most standards assign a criticality level to a system based on the severity of a potential catastrophe and the probability of its occurrence. These are then mapped onto categories for the system criticality levels. If software controls these safety critical systems, then the software too is assigned a criticality level. The software criticality levels correspond to the failure conditions that would result if the software were to fail.
The Federal Aviation Administration (FAA) recognizes five categories of failure conditions and five software-level definitions.

<table>
<thead>
<tr>
<th>Failure Condition</th>
<th>Software Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic</td>
<td>Level A</td>
</tr>
<tr>
<td>Hazardous/Severe – Major</td>
<td>Level B</td>
</tr>
<tr>
<td>Major</td>
<td>Level C</td>
</tr>
<tr>
<td>Minor</td>
<td>Level D</td>
</tr>
<tr>
<td>No Effect</td>
<td>Level E</td>
</tr>
</tbody>
</table>

Table 1: Categories of Failure

In practice, the differences between levels A and B are small:

- Certain objectives of the software design process must be independently verified.
- Source code accuracy, consistency, and compliance with the software architecture must be independently verified.
- Robustness of object code with low-level requirements must be independently verified.
- Test coverage of software structure (modified condition/decision) must be satisfied independently for level A, and is optional for level B and lower.

The Motor Industry Software Reliability Association (MISRA) classifications match integrity levels using five categories of controllability. [MISRA]

The new IEC 61508 standard uses four software integrity levels, and IEC 880 also uses four levels. [IEC 61508]
There is a strong correlation between the way software can contribute to a potential hazard and the severity level. The level assigned to some software has a great influence upon the rigor with which the software is developed and verified and the evidence that must be collected to confirm this.

**Standards**

**Avionics – RTCA / EUROCAE**

The avionics industry has taken the lead in the development of safety certification standards for computer programs.

The Radio Technical Commission for Aeronautics (RTCA), a nonprofit organization formed in 1935, has been instrumental in shaping the future of aviation through the application of electronics and telecommunications. The RTCA operates as a federal advisory committee that is composed of industry and government representatives. One of the key activities undertaken by the RTCA is the development and publication of guidance documents and minimum operational performance standards for avionics technology.

Although the acronym remains the same, RTCA now represents “Requirements and Technical Concepts for Aviation.” The RTCA board of directors and international associates work in association with members of the international aviation community to propose the formation of new committees and provide input to ongoing special committees.

One such committee, SC-167, was responsible for the preparation and revision of standards for the certification of avionics software. Major areas of concern include:
• Documentation
• Integration and production systems issues
• Software development
• Software verification
• Configuration management
• Quality assurance

SC-167 was responsible for the review and revision of the document *Software Considerations in Airborne Systems and Equipment Certification* (DO-178A). This review resulted in a new document, DO-178B, which was issued in December, 1992. The Federal Aviation Administration (FAA) of the U.S. Department of Transportation issued an Advisory Circular on January 11, 1993, stating that DO-178B may be used as a basis for submission of material required to obtain FAA approval of digital computer software. The document is known as ED-12B in Europe and is used by the Joint Aviation Administration (JAA) as a basis for certification.

As DO-178B was used by industry, it became apparent that parts of the document were unclear or ambiguous. A special committee was formed and tasked to produce guidance materials based on experiences and expertise in the industry. SC-190/WG52 is working on the production of these guidelines initially in four principal areas:

• Previously Developed Software –

  Examples of topics covered are: use of software developed to different standards; use of software developed as Commercial Off The Shelf (COTS) and
re-used in a safety critical application; use of software developed to a lower criticality level and subsequently used on a higher-level application.

• **Verification** –

  Topics include: the intent of structural coverage; use of high-level requirements to verify low-level requirements; definition of structural coverage terms, etc.

• **CNS/ATM** –

  DO-178B is an Airborne Safety Guideline. Ground-based systems (communication, navigation, surveillance, air traffic management) were not subject to the same safety guidelines in the past. As ground/air systems became more integrated, it was recognized that there would be additional benefit if the safety objectives and techniques for achieving them were harmonized. The ground-based community will use DO-178B for future projects, and additional guidance is under development.

• **Special Considerations** –

  A device used successfully on one aircraft type can be adapted to a different aircraft type. The history of successful use could raise the confidence of the device to a level where it could be trusted. The evidence required, the reverification guidelines, and the level of trust are the subject of additional guidance being developed to interpret the special considerations requirement.

  The committee continues with the development of the guidance documents, and the intent is that, once
approved, the recommendations will become an official interpretation of DO-178B.

ISO 9000

The ISO 9000 guidelines do not address the production of safety critical software, but rather focus on the issue of quality. All certification bodies insist on a quality system that instills confidence in the production of safety critical software. ISO 9000 standards are not sufficient in themselves to satisfy safety standards. However, any company undertaking the ISO 9000 certification process clearly makes a positive statement of intent regarding quality objectives.

Def-Stan 0055

The Procurement of Safety Related Software in Defence Equipment [DS 00-55] was published in August, 1997. It is composed of two parts. Part one covers the requirements and part two provides guidance. Like most others, this standard requires a thorough development and verification process. There is, however, a heavy emphasis on formal approaches to specification, design, verification, etc.

The Def-Stan 0055 standard recognizes that, although desirable, formal methods require special skills and tools, which may not be available to a project in a timely fashion. The representative program manager has a great deal of discretion on the use of formal methods, and how much they should be supplemented with alternative verification techniques.
Development Guidelines for Vehicle-Based Software

The Motor Industry Software Reliability Association (MISRA) is a consortium of automotive and component manufacturers, together with motor industry associations and a university. The MISRA members undertook to research specific issues relating to automotive software. This research resulted in nine detailed reports being published.

These development guidelines cover the software life cycle and offer a different perspective for the needs of the automobile industry. Factors in safety analysis must consider possible hazards associated with human interaction with a system. Driver experience, human reaction times, and attentiveness are some of the factors listed. Also included is the risk compensation factor, where improved safety can lead to more risky behavior.

Several of the reports discuss the use of languages and compilers:

Some safety-critical software pundits deprecate the use of ‘C’ due to its incomplete ISO definition resulting in many aspects of the language being undefined, unspecified or implementation specific. In these aspects, it is viewed as being weaker than assembler. The advent of C++ is regarded by some with horror as the language specification is even weaker than that for ‘C’ and elements of the compiler are becoming very complex.

Languages recommended for high-integrity applications are ISO Pascal subset, Ada subsets and Modula 2 subsets. — Report 1

... it is normally recommended that strongly typed and structured languages (e.g., Ada or Pascal) should be used, which provide compiler and run-time assistance in the
finding of faults, rather than the ‘flexible’ languages (e.g. C or Assembler) which require additional checks to be made by other means. — Report 2

Since the publication of the Development Guidelines for Vehicle Based Software, another document entitled Guidelines for the use of the C language in Vehicle Based Software has been published [MISRA-C]. In the introductory paragraphs the following statements are made:

Nonetheless, it should be recognised that there are other languages available which are in general **better suited** to safety critical systems

Examples of languages generally recognised to be more suitable than C are Ada and Modula 2. If such languages could be available for a proposed system then their use should be seriously considered in preference to C.

The MISRA C guidelines catalog language rules. There are 92 required rules and 35 advisory rules. Of the 92 required rules, 70 of them do not apply to Ada. Of the 35 advisory rules, 27 of them do not apply to Ada. An example of a rule that applies to both C and Ada is “no-recursion.” An example of a rule that does not apply to Ada is “do not use leading zeros in numbers to denote octal-based numbers.”

Many of the problems addressed by the C guidelines, which require special care and attention by the C programmer, are solved by the Ada language definition and implementation.

**IEC 880**

In 1986, the International Electrotechnical Commission (IEC) published a standard on *Software for Computers in the Safety Systems of Nuclear Power Stations*. This IEC 880 standard is applicable to the highly reliable software
required for computers used in the safety systems of nuclear power plants for safety functions.

Although no specific language is recommended, guidance is given for the selection of a suitable language based on some common basic rules for safety-system programming languages. For example:

- Languages with a thoroughly tested translator
- The language must be completely and unambiguously defined
- Problem-oriented languages are strongly preferred
- The language should not prohibit:
  - Error-limiting constructs
  - Translation-time type checking
  - Run-time type and array-bound check, and parameter checking

A safety critical subset of Ada satisfies all of these criteria.
Use of Ada in Safety Critical Systems

Use of Ada in safety critical systems affords many software architecture characteristics, making it an ideal choice for such systems. Aeronautical Radio, Inc. (ARINC) is an organization in which the U.S. scheduled airlines, air transport companies, aircraft manufacturers, and other non-U.S. airlines are stockholders. ARINC Report 613, *Guidance for Using the Ada Programming Language in Avionic Systems*, makes the following statement:

> It is the desire of the airline community to reduce the cost and the economic risk associated with avionics software systems. As a means to achieve this, it is recommended that the Ada programming language be used as the standard High-Order Language (HOL) in avionics equipment design.

ARINC Report 613 also makes the following recommendation on the certification of the Run-Time Library (RTL):

> It should be the compiler manufacturer's responsibility to verify the RTL, using the guidelines of DO-178A to the highest level of criticality . . .

The Boeing Commercial Airplane Group (BCAG) has selected Ada as the programming language for use in digital avionics computers. Ada was used by Boeing and by their suppliers to develop code for the new Boeing 777 aircraft. Future commercial aircraft and systems will also use Ada. *BCAG Airborne Software Product Requirements [D6-81925]* specifies:

> For software levels A or B the supplier shall implement source code in a programming language as defined in table 2.1-1. Table 2.1-1 lists just two languages: *Ada 83* and *Ada 95*. 


The [D6-53339] document lists requirements that specify the features of the Ada language and how they may be used in digital avionics software. The document also addresses the Ada Run-Time System (RTS) and states the following:

3.2 For the purposes of airborne digital computer avionics systems, the Ada RTS is part of the software end item. The following apply if an Ada RTS is used:

a. The features of the Ada RTS loaded into the software end item shall be verified to the same verification type as the software application itself.

b. The supplier shall provide evidence to BCAG and the regulatory agency that the Ada RTS is deterministic and meets certification requirements.

The supplier shall select a compiler and RTS from a vendor who will support certification of the RTS. This means that the vendor will provide data on the development and verification of the RTS. ...

These and other safety standards require or recommend the use of industry-best practices in all aspects of systems development. One area of key importance is the programming language used as the basis of the final installed system. The standards specify the use of a language which is well defined, has validated tools, enables modular programming, has strong checking properties, and is clearly readable.
The Benefits of Ada in Safety Critical Systems

Ada has properties that make it a natural choice for the development of safety critical systems.

- **Ada is an International Standards Organization (ISO) standard**
  
  It is well defined, stable, and provides a portable foundation for the development of supporting tools and libraries. Its well established validation mechanism, Ada Compiler Assessment Test Suite (ACATS), ensures trust in the general quality of Ada compilers.

- **Ada supports Object Oriented Programming (OOP) and Object Oriented Design (OOD)**
  
  In particular, there is strong support for abstraction and the re-use of tested components.

- **Ada has a legible style**
  
  This is important for the satisfactory execution of certification steps such as peer review and walkthroughs as well as for later maintenance.

- **Ada has a coherent modular construction**
  
  Separate compilation facilities enable applications to be written as a set of units, with interface specifications and the dependencies on their use clearly exposed. Ada compilers enforce these dependencies and ensure that, if any interface specification is changed, then the corresponding unit using the interface shall also be recompiled. This process enables the organized construction of a program from trusted components.
• Ada encourages the detection of errors at an early stage of development

Strong typing facilities enable the user to construct programs in which the correct use of data is controlled by its declaration. Properly used, this ensures that most errors are detected statically by the compiler and that many remaining errors are automatically detected at execution.

• Ada contains low-level features allowing basic elements of the target hardware to be accessed in a logical manner

The address representation clause, enumeration representation clause, and unchecked conversions are some of the features that enable a program to be mapped to the target processor directly.

• Ada provides control over the visibility of types, operations, and data, thus limiting the features that may be used by any program unit

For example, before the generic function Ada.Unchecked_Conversion can be used, it must be made visible by a with clause. This exposes the places in which this potentially unsafe feature might be used, allowing special treatment and testing to ensure that the safety of the overall program is not compromised.

It is instructional to note that the C language is criticized in this regard by the IEC draft standard on safety software [IEC/SC65A 91]. “C includes 150 undefined 'features' which severely limits its use in safety critical systems.” [Rat 95].
A study undertaken by A. Hill [Hill 91] of Nuclear Electric states the following:

The languages C and Ada are compared in some detail, from the point of view of their suitability for use in safety critical software systems or indeed for any application for which reliability is a major concern. The conclusion reached is that C is unsuitable because it contains a large number of dangerous features. Ada, on the other hand, contains many features which safeguard program integrity and is believed to be wholly appropriate for this purpose.

The conclusion is clear – of all the widely available languages suitable for safety critical software, only Ada is an appropriate baseline for these applications.
Restricting Ada for Safety Critical Systems

It should be noted that Ada is a general-purpose language and contains a number of language features that should not be used in safety critical systems. A safety critical program must be totally bounded in time and the amount of memory used. The time taken to execute and the memory used by the operational software must be determined and verified as part of the certification process. It is extremely difficult to determine the bounds of certain Ada constructs, as they include calls to the run-time system, which uses run-time data structures that change as the program executes.

Tasking

Certain tasking operations require calls to run-time routines, which may scan and sort various queues and analyze several data structures during the scheduling process. The time taken to perform these operations depends on the state of the tasking queues, which in turn depends on the state of the tasks at any given time.

When the combination of these states becomes complex, it becomes very difficult to predict the exact operational sequences that will take place. Writing a set of tests for these tasking operations is formidable. The current state of the art precludes such complex and rigorous testing. Use of the full features of the Ada tasking system is not recommended in safety critical systems.

A subset of the tasking system, which constrains tasking operations to a deterministic subset, has been defined and is described in the Ravenscar Profile section on page 20.
Exception Declarations

The time and memory used during the elaboration of exception declarations and during the raising of an exception are predictable. Finding an exception handler, once an exception has been raised, involves searching through subprogram stack frames, or looking through tables that hold exception handler addresses.

The time taken to locate the appropriate handler depends on the dynamic nesting of subprograms. Testing for all possible subprogram combinations at each point at which an exception can be raised presents a combinatorial explosion of states, since exceptions can be raised implicitly during program execution. When an exception is raised, control is transferred from any statement in the enclosing frame, which makes it difficult to predict the state of the global variables. Thus, the use of nested exception handlers is to be avoided in safety critical systems.

Other Restrictions

Use of certain language constructs requires support code and data structures, which make verification difficult. Data structures that require the use of heap memory are forbidden since the support code must deal with complex memory management operations whose execution times are unpredictable. Operations, which require searches, and data, which results in multiple access paths, are constructs that are restricted in safety critical software. The guidance document produced by the HRG addresses the use of such Ada constructs in high integrity systems.
Run-Time Requirements

We have seen that, although Ada provides an excellent foundation for safety critical systems, certain features must be avoided. The key lies in the run-time system or real-time executive, which contains the routines required to support those parts of the language that cannot be implemented directly with inline code. Ada programs implicitly require run-time system support during program execution.

The run-time system must be provided in the program library, which is used during the compilation of an application. While application developers have control and visibility over their own code, the Ada run-time system is usually not visible to users. It may be possible to limit the program constructs to a subset that requires no run-time support at all or to inline the run-time support code. This puts the burden of programming and verification on the user.

Since the run-time system is part of the delivered executable program for a safety critical application, it must be subjected to the same hazard analysis and certification as the application code. Consequently, as part of the deliverables, full documentation and certification materials must be supplied, not only for the application code itself but also for the run-time system actually used.

The normal run-time system for full Ada is not appropriate for safety critical systems (which use only a subset of the language) because it contains code that uses techniques outside the certifiable regime.
Ravenscar Profile

The International Real-Time Ada Workshop (IRTAW) met in April, 1997, in Raven Hall (Ravenscar – UK). The purpose of the workshop was to discuss and develop a subset model of an Ada 95 run-time system. The characteristics of this model are:

- Small size
- Fast performance
- Predictable behavior

It was recognized that this model would support the subset of the language important to the developers of “hard real-time,” secure, and safety critical applications. An implementation using the Ravenscar model will not permit unbounded language features (for example, nested tasks, dynamic heap management, and complex task synchronization).

The Ravenscar model is defined by the following:

- All tasks and protected objects (POs) are defined at the library level
- No dynamic creation of tasks or POs
- Static, fixed maximum number of tasks
- No task entries (therefore, no rendezvous)
- Protected subprograms and protected entries supported
- One protected entry per PO
- Maximum of one caller queued on entry at any time
- Tasks do not terminate (they are all global anyway; knowing this reduces overhead)
• Task discriminants supported
• Task_Id supported
• Dynamic_Priorities package (optional)
• Tasking operations supported:
  - Delay until Real_Time.Time
  - Protected entry call
• No protected operations except E'Count outside of barrier expressions
• Barrier expression on a protected entry must be a boolean variable

Consequently, the following are excluded:
• Abort, delay (relative), package Ada.Calendar, all forms of rendezvous, asynchronous transfer of control, requeue, timed and conditional protected entry call, Task_Attributes package, 'Callable, 'Terminated, 'Count (except 0 or 1 for PO).
• Controlled types are so there is no requirement for finalization.

Although the model constrains the scheduling options, it is pre-emptive and supports Rate Monotonic Analysis (RMA).

The Ada tasking system includes a rich set of features, some of which are inappropriate for safety critical systems. Certain tasking constructs must be avoided and a style of programming imposed which ensures deterministic behavior.

**Cyclic Executives**

Traditionally, safety critical real-time control systems are programmed using simple cyclic executives. These are usu-
ally implemented as a loop, which calls procedures in turn. A simple representation of such a loop is shown in the figure below:

![Loop Representation Diagram](image)

**Figure 1: Loop Representation**

The execution sequence of such a loop continues calling procedures $P_a$, $P_b$, and so on, in turn, and each procedure executes and returns control back to the main loop.

The calls are made in a cyclic sequence. When the execution of the last procedure completes, a clock function may be invoked to delay the call of $P_a$ until a known time-point, such that each call of $P_a$ starts at a fixed time interval established by the clock. An alternative mechanism may be to suspend the main program after $P_h$ (perhaps executing some background task) and delaying the restart of the loop until trig-
triggered by an interrupt generated externally at a prescribed
time interval. This time interval is known as the time frame.

It is essential that all of the procedures, P_a through P_h, com-
plete their actions within the specified time frame.

In safety critical systems, the longest path through each pro-
cedure must be estimated and verified through test. When
taken in aggregate, the sum determines if the work required
can be completed within the allotted time frame.

There are a number of problems with this approach:

1. All of the procedures are executed with the same
   frequency and at the same priority. They follow a strict
   sequence defined by the loop. To increase the frequency
   of some procedures, a nested loop may be invoked,
   which calls the procedures several times. This
   complicates the timing analysis; so it is unusual to have
   more than two loops, one nested within another. They
   are usually called the major and the minor loop.

2. Request for input is typically handled through one or
   more of the procedures. Various inputs arrive as
   asynchronous events because they occur at different
   times. The information is deposited in various global
   locations, which are then polled in a strict sequence
   defined by the procedures in the loop. The rate and
   sequence of data consumption is not governed by the
   natural occurrence of the data, but by the characteristics
   of the loop.

3. The various procedures around the loop obtain
   information from global locations that compute results
   and deposit them in global locations. The transformation
sequences are driven by the structure of the loop rather than by the actual requirements on the data transformations themselves.

4. Once the program is developed and tested, any subsequent changes require the measurements and analysis to ensure that the loop still performs within the time frame allotted. The programs are thus driven by the execution time frames rather than by the event times.

The alternative is to choose a different programming paradigm, one in which the code of these procedures may be scheduled independently.

**Periodic Tasks**

Rather than having one loop which calls the procedures, each of these procedures may be mapped to a task, with the body of the task implemented as a loop. The main loop of the task contains the same code as in the procedure $P_a$, but at the end of the loop a `delay until` statement causes the task to be suspended until it needs to be run again. Since these tasks execute continuously in a loop at a frequency that is determined by the `delay until` statement, they are called periodic tasks.
An example of the structure of a periodic task is shown below:

```vhls
task Ta is
    pragma Priority (10);
end Ta;

task body Ta is
begin
    loop
        ....
        -- processing from body of Pa
        ....
        Next_Frame := Next_Frame + Frame_Time;
        delay until Next_Frame;
    end loop;
end Ta;
```

To implement the same cyclic scheduling code using tasks instead of procedures $P_a$ to $P_h$, each procedure is mapped to a corresponding task, $T_a$ to $T_h$. Each of these eight tasks is given a different priority, with task $T_a$ at the highest priority, $T_b$ to be at the next highest, and so on down to the lowest priority task, $T_h$.

Task $T_a$ completes its processing, calculates $Next\_Frame$, and delays until that time arrives. Task $T_h$ then completes its processing and delays until $Next\_Frame$. Tasks $T_c$ to $T_h$ follow in turn. The effect is that all tasks execute just once in each time frame and then are suspended until the end of the current frame. At the start of each time frame, all tasks are ready to run, in priority order. If each of the tasks suspends itself until the end of the frame, then this implements the cyclic executive scheduling algorithm.
Each task may be run at its optimum frequency subject to the availability of hardware resources. If the time frame for these tasks is 50 milliseconds, then a \texttt{delay until} statement in each task increments the time of the next frame by this amount in each loop.

The schedulability of the system can be determined by finding the maximum time taken for each task in the periodic loop and dividing this by the period in which each task has to be repeated. These ratios define the portion of time that the processor must devote to each task. The sum of these ratios may be used to calculate and check if there is sufficient processor time available to ensure that no task misses its deadline.

So far, only periodic tasks have been considered. These run concurrently and do not interact. Notice, however, that the period of each task may be different and should reflect the repetition rate required by the application rather than the rate imposed by the program structure.

**Event Driven Tasks**

The full Ada language provides several mechanisms for tasks to synchronize operations. There are three main reasons why tasks need to synchronize:

- To perform some operations in response to an external event (an interrupt)
- To sequence some operations so that they occur when an agreed-to condition is satisfied (an event)
- To protect access/modification of data structures to ensure that the operations are consistent in the presence of concurrency
The Ada language provides the rendezvous construct to support these needs. The Ravenscar Profile prohibits the use of the rendezvous and suggests the use of the protected object (PO) instead. PO’s provide a more primitive, but much more efficient mechanism to synchronize tasks and control access to shared data. The Ravenscar Profile further restricts PO’s to ensure that they have no need for additional queueing mechanisms in their implementation.

Protected procedures may be attached to interrupt handlers. An interrupt may simply result in some code being executed in response to the hardware event, or the handler may trigger a task awaiting a corresponding software event.

An alternative synchronization mechanism called Synchronous_Task_Control is defined in Annex D (Real-Time Systems Annex) of the Ada Reference Manual (ARM). This package allows users to define very simple suspension objects with their corresponding operations. The simple Suspend_Until_True, Set_True, and Set.False operations provide the primitive mechanisms to synchronize operations between tasks. Although not explicitly mentioned in the Ravenscar Profile, the implementation of this package is not precluded. It provides a simple building block to construct higher level application specific mechanisms.

**Benefits of the Ravenscar Profile**

The Ravenscar Profile possesses the following advantageous software features:

- The run-time support code and the corresponding data structures are small
- Timing behavior is predictable
- Analysis is based on a sound theoretical foundation
• Code/data coupling is directly visible
• Sensitivity to timing changes is constrained
• Flexibility is increased because:
  - Work done by any task can be changed
  - Periodicity of any task can be changed
  - Analysis will show if schedulability is achieved
Ada 95 for Safety Critical Systems

Annex H – Safety and Security

The ANSI/ISO standard for Ada [Ada 95] addresses the requirements for systems that are safety critical. The language is described in the core document, required libraries are defined in Annexes A and B, and optional packages and pragmas are described in annexes C through H.

The safety and security annex, Annex H, covers three topics:

- Understanding program execution
- Reviewing object code
- Restricting language constructs whose usage might complicate the demonstration of program correctness

These features are introduced through compiler directives, called pragmas, as defined in Annex H. Although these pragmas are a useful addition to the Ada language, they are not required to help obtain safety certification for an Ada program.
These extensions address the needs of certification through test and review, and also provide for the inclusion of formal verification and validation tools.

**HRG**

The Annex H Rapporteurs Group (ISO/IEC/SC9/WG22/HRG) was formed to produce a guidance document for the use of Ada 95 in high integrity systems. Over a three year period, the group produced a document which is currently under review and on track to becoming an ISO/IEC technical paper [ISO/IEC PDTR 15942]. The group analyzed several sector-specific standards and extracted the requirements for software verification.

<table>
<thead>
<tr>
<th><strong>Pragma</strong></th>
<th><strong>Effect</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizeScalars</td>
<td>Enforce an implementation convention which sets uninitialized variables to a predictable value. (outside the range, if possible)</td>
</tr>
<tr>
<td>Reviewable</td>
<td>Complete without aggressive optimizations enabled. Document the mapping to the object code.</td>
</tr>
<tr>
<td>InspectionPoint</td>
<td>Compile without aggressive optimizations enabled. Generate memory mapping information.</td>
</tr>
<tr>
<td>Restrictions</td>
<td>Enforce user-selected safe subset of the language.</td>
</tr>
</tbody>
</table>

Table 2: Pragmas
Ada 95 constructs were then analyzed and a set of tables produced which make a statement on the degree of difficulty that a particular construct or construct combinations present to various verification techniques.

The guideline does not make recommendations on the constructs that could be used and those that should be avoided, as this depends on the safety guidelines, the certification level, and the verification methods proposed. Safety critical projects could use this information to specify the design and coding standards specifically for the project in combination with the proposed verification plans.
Certification

The safety certification of an application, along with the executive embedded within it, consists of verifying that the construction and testing were performed according to a set of rigorous software engineering and safety standards. The avionics guidelines document DO-178B includes some strict requirements.

What is Required for Certification

All certification guidelines stress the importance of a process based on sound engineering practices. The steps to be taken in the development of safety critical software must be well understood and documented before the software can be certified. Rather than waiting until the software is fully developed and tested, it is wise to involve the certification authorities in the early planning stages.

Planning Documents

To raise the confidence in safety critical software, the development stages used in its production must be understood. Software developed by a software engineer working alone does not instill the confidence required to certify a system for flight. A preferred approach is to use a team that follows a controlled software engineering method. It is important that this method be used consistently throughout the project.

Plan for Software Aspects of Certification (PSAC)

This is one of the first documents to be produced, laying the foundation for the production of a safe system. The PSAC addresses the following areas:

- System overview
• Software overview
• Certification consideration
• Software life cycle
• Software life cycle data
• Scheduling

Software Development Plan (SDP)

This plan must describe all development stages as well as describing the materials developed and their acceptance criteria, including:

• Standards
• Software life cycle
• Software development environment

The software development process usually proceeds according to the cascading sequence of events, as shown in the standard “Waterfall Model” on page 34.
Figure 2: Software Development Process - “Waterfall Model”
Software Verification Plan (SVP)

This plan describes how the evidence to be used for safety assessment shall be collected and presented. Verification of system safety is done by review, testing, and sometimes formal analysis. The plan must show how confidence is built up in the lower-level software components and how the integration of these components satisfies the requirements of the system as a whole. Although thorough testing is always required, there is growing interest in the use of formal verification methods at the source level.

The following topics must be addressed in the SVP:

- Organization
- Independence
- Verification methods
- Verification environment
- Transition criteria
- Partitioning considerations
- Compiler assumptions
- Reverification guidelines
- Previously developed software
- Multiple-version dissimilar software

Software Configuration Management Plan (SCMP)

This plan describes the environment and the activities required to produce, maintain, and record data during the project life cycle.
Software Quality Assurance Plan (SQAP)

This plan describes the verification of life cycle processes and data, along with activities and procedures conducted by SQA staff, to show evidence of records and data that support independence.

Standards

Software Requirements Standards

This standard defines the method, rules, and tools used to develop high-level software requirements.

Software Design Standards

This standard describes the methods, rules and tools used to develop the software architecture and low-level requirements.

Software Code Standards

This standard defines the programming language, methods, rules, and tools used to code the software.

Data Documents

The following documents serve as the repository for data and records maintained for lifecycle processes.

- Software requirements data
- Design description
- Source code
- Executable object code
- Software verification results
- Software verification cases and procedures
• Software life cycle environment configuration index
• Software Configuration Index (SCI)
• Problem reports
• Software configuration management records
• Software quality assurance records

Testing

Several kinds of testing strategies are required to achieve confidence in a safety critical system. “Black Box” testing checks that each function generates the expected results under all conditions that the function might encounter. Each function must be tested with its typical data values, and also with its data values at the boundaries to check the extreme conditions that could be experienced.

“White Box” testing (also known as “Glass Box” testing) is a more stringent testing process. It involves analyzing the structure of a function under test to ensure that all the elements of the function are required, all the elements are executed, and all execution paths in the application are adequately covered. The tests must ensure that the program executes all conditions, and that all conditions work correctly when evaluated to both true and false.

The tests and testing environment must be designed to ensure that the tested software is as close to the final configuration as possible. If the testing environment is intrusive, the test results must describe the expected differences between the tests and the final product.
Condition/Decision Coverage Testing

Conditions are boolean expressions which contain no boolean operations. A decision may contain several conditions which are combined with boolean operators, as shown in the following example:

\[
\text{if } A=B \text{ and } (C2 \text{ or } D>3) \text{ then}
\]

Figure 3: Identification of Conditions and Decisions.

If several conditions are combined with boolean operations, then this becomes a decision. For DO-178B the requirements for coverage testing specify that:

- All decisions must be executed, with both possible outcomes executed at least once
- Every condition in a decision must take on both outcomes at least once

Assigning a boolean expression to a boolean variable and testing the variable does not reduce the combination of
tests. The same coverage tests would be required for the following code:

```plaintext
TEMP := A=B and (C2 or D<3);
if TEMP then
...
```

Modified Condition/Decision Coverage

Modified condition/decision tests are more difficult to formulate. MCDC tests show that each condition will independently affect the outcome of the decision. Tests are written in combinations such that for each condition there is a pair of states that change only the one condition and there is a corresponding change in the decision outcome.

For the Ada statement:

```plaintext
if A=B and (C2 or D>3) then P; end if;
```

the following tests must be shown to comply with these truth table results:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A=B</td>
<td>C2</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 3: Condition/Decision Tests
In the table above, condition:

A=B – shows corresponding results in cases 3 and 4
C2 – shows corresponding results in cases 1 and 2
D>3 – shows corresponding results in cases 1 and 3

One way to avoid modified/condition decision coverage is to use the Ada short-circuit conditions:

```ada
if A=0 and then B<2 and then C>5 then
    P;
end if;
```

This ensures that the conditions are only evaluated so far as they are needed to achieve the final result. The short circuit form is equivalent to three nested `if` statements:

```ada
if A=0 then
    if B<2 then
        if C>5 then
            P;
        end if;
    end if;
end if;
```

However, this shortcut method of avoiding MDCD coverage testing is only valid if the testing method shows each condition being tested to TRUE and FALSE, and not just the decision.

Coverage Testing

Operational software (software that is loaded as part of the application itself) must satisfy the requirements of the system. *There must be no additional software that cannot be traced back to requirements.* The rigor with which this is
checked depends on the criticality of the application. Under the DO-178B guidelines, level B requires the relationship between requirements and code to be verified at the source code level. At level A this must be shown at the object code level, unless the traceability between source and object code is verified.

Compilers usually contain complex algorithms that use data and code flow information to optimize the use of machine registers. Inserting code to trace the results of conditions and decisions will affect the registers used and possibly the order of evaluation. Some method of mitigating the risk of missing some condition during test is required if source is modified to obtain coverage. One approach may be to compile and test with different compilers and verify equivalent coverage results.

**Control to Data Coupling**

DO-178B requires that the relationship between data and the code that transforms it is documented. A flight control system implemented using a cyclic scheduler might perform the following:

- Obtain input from the pilot’s control stick
- Transform the position of the stick into a request to change direction
- Perform control law calculations
- Transform request for flap position into actuator signal
- Send signal to actuator

In this very simplified set of steps, the data is passed and transformed through many code functions between input and
output. Data-to-code coupling shows that the data values are passed to the intended functions. The objective of this analysis is to show that a change in a data value does not change another value unintentionally. For example, a movement of the flight control stick must not cause the wheels to change position.

Systems that pass information through unstructured global data areas make data-to-code coupling analysis difficult. Ada has features that can make implementation, documentation, and subsequent analysis much easier. Putting type and data information in package specifications, use of private types, use of parameter modes and so on, helps to establish the relationship between data and code that uses the data. Careful use of scope, visibility, and access rules can make the data-to-code coupling obvious.

Data “Freshness”

In a cyclic scheduler, data input is obtained by polling some value at the repetition rate for the loop. The actual data arrival rate may vary. For example, an analog/digital converter, which counts incremental voltage steps, takes longer to convert a higher voltage than a lower one. It could be difficult to couple data values to the particular invocations of the functions that process it.

Control systems that require such coupling to be shown may need to increase processing power to ensure that the loop is always “ahead” of the data, improve the quality of the data input device, downgrade the performance of the system, or take a number of other steps to ensure data integrity.
Event-Driven Solution

It may be easier to certify an event-driven system than a cyclic-based design (except for the simplest cyclic applications).

A protected object provides a natural and direct coupling between a data value and the code that uses it. Tasks may be triggered to process data on its arrival. Such event-driven systems implement control based on system needs rather than on the flow imposed by the code structure.

Event-based scheduling, together with well structured scope and visibility controls, provide an easier way to demonstrate data to code coupling.

Software Accomplishment Summary (SAS)

All of the system and software requirements must be adequately covered by tests. Implicit derived requirements, including initialization of the stack or set-up of heap addressing registers, must also be tested. To ensure that every requirement and every byte of code is tested, a compliance matrix must be produced that records the relationships between documents, code, tests, and test results. The matrix must be identified in the SAS.

The entire development process used for the project is recorded in the SAS, which provides documentation of compliance with the PSAC.
Conclusions

The avionics industry has for some time recognized the importance of software in safety critical systems and has developed standards for ensuring that the software in such systems meet appropriate levels of reliability. Society, as a whole, is now beginning to realize that software is pervasive and that our lives and property depend upon software to an astonishing degree. Other industries are following the lead of the avionics community by developing their own standards for the creation of reliable software.

Ada plays an important role in the development of safety critical software because it is the only commercially available language with characteristics corresponding to various standards, such as those developed by the avionics industry. The Ada program ultimately delivered requires an embedded run-time system as well as project-specific Ada code.

Liability Considerations

On July 25, 1985, the European Community (EC) Council ratified a Directive Concerning Liability for Defective Products 85/374. Under this directive, products are considered to be defective when they do not provide the level of safety that the public has a right to expect. The directive creates a strict liability factor and introduces a uniform concept of product liability in some EC nations where such a view did not previously exist.

The EC Directive on General Product Safety 92/59 was approved by the EC Council. The directive became fully operational in June, 1994, and applies to all products placed on the EC markets. The objective of the directive is to impose a general requirement on producers to introduce only
safe products into the EC market. Some legal opinion has been expressed that appropriate quality standards (ISO 9000) together with safety certification materials could be useful in a legal defense.

**Trends**

The era of safety critical software is just beginning. Software applications will increase in size and complexity as the move toward automated systems continues to grow in all sectors. Public expectations for safety in products and services will also increase. The wide acceptance of the automotive industry's introduction of air bags and anti-lock brakes is clear proof that customers will pay a premium for safety features.

As product safety becomes more of an international imperative, a growing number of industries will be forced to develop and enforce their own standards for safety critical certification. However, this process can also result in a number of positive benefits to the enterprise:

- Products and services will be safer and more reliable.
- Catastrophic financial liabilities resulting from lawsuits and product failures or recalls can be minimized.
- The market advantage of using “certified” software can result in greater sales and profitability.
Recommendations for Safety

A few simple precautions can be taken by manufacturers to safeguard against the risks of software failure and its related liability:

- Be sure that software is included in the hazard analysis of any system under specification.
- Ensure that the degree of safety analysis for software is as thorough as that for the rest of the system.
- Build safety requirements into the application from the start.
- Adopt an agreed-to set of standards and procedures that satisfy the requirements of the certification authorities.
- Ensure that the staff employed have the relevant knowledge and experience to develop and verify a safety critical system.
- Use an implementation language that supports the rigor required for safety critical software, namely Ada.
References


[also the European joint standard: EUROCAE ED-12A, December 1992.]


This page is left blank intentionally.