BSSC(98)3 Issue 1
October 1998

Ada Coding Standard

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: BSSC(98)3			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1998	

Approved, November 18th 1998
Board for Software Standardisation and Control
M. Jones and U. Mortensen, BSSC co-chairmen

TABLE OF CONTENTS

PREFACE

This Ada Coding Standard is based upon the experience of developing custom space system software using the Ada programming language. This standard replaces the Draft Ada Coding Standards issued by the BSSC in 1990 as BSSC(87)2 Issue 1.

The BSSC wishes to thank Tullio Vardanega and Richard Jack for preparing the standard. The authors wish to thank all the people who contributed ideas for this standard.

Requests for clarifications, change proposals or any other comment concerning this standard should be addressed to:

BSSC/ESOC Secretariat
Attention of Mr M Jones
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr U Mortensen
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

This page is intentionally left blank

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

This purpose of this document is to specify a coding standard for programs written in the Ada programming language. The document proposes a set of stylistic guidelines designed to aid the production of high-quality, readable, reliable, maintainable and reusable Ada programs.

The document may equally serve as a self-standing standard or as the basis for a more comprehensive company-wide or project-specific standard.

The coding standard proposed in this document applies to any program written in the Ada programming language, irrespective of the specific application domain.

Effective application of the present guidelines to specific projects and application domains (such as, for example, those applying to on-board application as opposed to ground application) may require the generation of domain-specific complements. The formulation of domain-specific coding practices and recommendations, though, falls outside the scope of this document.

1.2 OVERVIEW

Chapter 2 provides an overview of the evolution of the standards for the Ada programming language. Chapter 3 describes the practices for presentation of code. Chapter 4 discusses program structure techniques to aid clarity. Chapter 5 discusses good programming practices. Chapter 6 discusses the issues affecting portability of Ada programs. Appendix A contains code examples using the relevant coding standards. Appendix B contains a list of references used in the text.

This page is intentionally left blank

CHAPTER 2
THE ADA PROGRAMMING LANGUAGE

2.1 INTRODUCTION

The Ada programming language, which was first defined as ANSI/MIL-STD-1815A, has been specified as an ISO (International Standardisation Organisation) standard since 1987 [Ref. 1]. This language standard has since been the subject of an extensive revision process which has led to [Ref. 2].

The language standard in its former version is commonly known as Ada 83 whereas the language standard in its latest version is referred to as Ada 95.

The ISO body in charge of the language revision has defined the intended transition process from Ada 83 to Ada 95. The transition process foresees a limited period of time (1995-1998) during which both language versions would coexist and both conformance certification suites (known as ACVCs) would be maintained.

At the end of the transition period, the old Ada standard shall be declared obsolete and all new certificates of conformance shall solely address the revised Ada standard.

An ISO working group to which ESA is actively participating has prepared a guidance document to the production of High-Integrity Ada Systems [Ref. 3]. This guidance document should constitute one major complement to the present coding standard.

2.2 TRANSITION ISSUES

The issues of transition from the old to the new language standard fall into two main categories: the incompatibilities between the languages and the exploitation of the new language features.

Upward compatibility of Ada 95 was one of the major design goals of the revision process. The goal was achieved in that the persistent incompatibilities between Ada 83 and Ada 95 are very limited in number and extent and also easily overcome. The reader is referred to the Ada 95 Rationale (Appendix X entitled Upward Compatibility) [Ref. 5] for a comprehensive discussion of such issues.

The main new features and enhancements contained in the Ada 95 language standard are summarised as follows:

1. object-oriented features
   - tagged types
   - controlled types
   - polymorphism
   - multiple inheritance
2. abstract types and subprograms
   - program structure and compilation
   - child library units
   - generics
3. refinement of tasking model
   - protected types
   - new forms of synchronisation mechanisms
4. additions to declarations and types
   - access-to-subprogram types
   - refined access types
5. provision of specialised annexes
   - system programming
   - real-time systems
   - distributed systems
   - information systems
   - numerics
   - safety and security

2.3 LIMITATIONS OF CURRENT STANDARD

The core of the present Ada coding standard is drawn from the

organisation and contents of the AJPO's Ada Quality and Style document [Ref. 5] which is presently recognised as the single most authoritative source of coding guidelines for Ada programmers.

The provision of coding prescriptions and recommendations regarding the usage of the new features and enhancements of Ada 95 is, for the moment, excluded from the scope of the present coding standard in view of the relatively limited maturity and the to-date insufficient utilisation experience.

This page is intentionally left blank.

CHAPTER 3
SOURCE CODE PRESENTATION

3.1 CODE FORMATTING

Code layout affects the look of the source code and not its operation. The adoption of a consistent code formatting style simplifies the development and maintenance of the source code.

The code layout style adopted in the Ada Language Reference Manual (LRM) [Ref. 2] shall be considered as the primary reference basis for any specific code formatting style. There are tools that exist within and without Ada compilation systems that support automated reformatting of Ada source code in accordance with the LRM best practice. These should be used whenever possible.

The code formatting conventions to be established as part of a company- or project-wide coding standard shall provide stylistic rules for:

- horizontal spacing
- indentation and alignment
- line length.

The practices are defined in the table below. In this and subsequent tables of practices, numbered supporting examples are found in Appendix A.

Id	Practice	Supporting Example
001	Consistent spacing around delimiters shall be used.	
002	The recommended paragraphing shown in the LRM for structure constructs shall be used.	
003	Indentation and alignment of nested control structures, using a three-space basic step of indentation, shall be used.	Example 1.

Id	Practice	Supporting Example
004	Indentation and alignment of continuation lines using a two-space basic step of indentation shall be used.	
005	Indentation and alignment of bodies of program units using a three-space basic step of indentation shall be used.	
006	Operators shall be aligned vertically.	Example 2.
007	One declaration per line, at most, shall be used.	
008	All declarations in a single declarative part shall be indented at the same level.	
009	Declarations shall be aligned vertically.	
010	Parameter modes and parentheses shall be aligned vertically.	Example 3.
011	Blank lines shall be used to group logically related lines of text and to separate them out from surrounding text.	Example 4.
012	Each new statement shall be started on a new line.	
013	No more than one statement per line shall be written.	
014	Compound statements shall be broken over multiple lines.	
015	The length of source code lines shall be limited to a maximum of 72 characters.	
016	The adopted code formatting conventions shall be consistently applied throughout each and every compilation unit produced as part of the program.	

3.2 NAMING CONVENTIONS

Consistent use of naming conventions improves program readability. Several naming conventions may be defined to differentiate the entities (objects and types) composing a program.

This standard does not attempt to instigate any particular naming convention. The establishment of a comprehensive naming convention needs to call upon several factors (such as, for example, linguistic practices and domain-specific terminology) which can hardly be standardised across boundaries.

This standard confines itself to instigating the basic principles for any such conventions.

The practices are defined in the table below.

Id	Practice	Supporting Example
017	Visually distinct naming conventions for Ada reserved words and other elements of the program shall be used. Note: Use `lowercase` names for Ada reserved words and `Capitalised` names for identifiers.	
018	Underscores to separate words in a compound name shall be used.	
019	The use of acronyms shall be avoided. Names that are as self-documenting as possible shall be used.	
020	The coining of new abbreviations shall be avoided. Only well-known, unambiguous abbreviations shall be used and, even then, they shall be used sparingly.	
021	Only singular, general nouns as type, subtype and record component identifiers shall be used.	
022	Only singular, specific nouns as object identifiers and non-Boolean functions shall be used.	

Id	Practice	Supporting Example
023	Predicate clauses or adjectives for Boolean objects and Boolean functions shall be used.	
024	Action verbs for procedures and entries shall be used.	
025	Packages shall be named using nouns that describe the abstraction provided.	
026	Tasks names shall describe the performed activity.	

3.3 NUMERIC LITERALS AND NAMED NUMBERS

The requirements are defined in the table below.

Id	Requirement	Supporting Example
027	Ada offers numeric literals and based literals to express numeric information. A representation appropriate to the problem shall be chosen and used consistently across the entire program.	
028	Underscores shall be used to separate the logical units of the chosen representation format in the expression of literals.	Example 5.
029	Based literals shall be used in the place of numeric literals where the resulting expression is more expressive of the origin or purpose of the literal value.	Example 6.
030	When using exponent notation in literals, the uppercase or lowercase notation shall be applied consistently across the entire program.	
031	Ada supports the declaration and use of named numbers. Named numbers denote the value of an associated static expression which is	

Id	Requirement	Supporting Example
	evaluated at compile time with full precision. Hence, named numbers should be used as the target of all the static numeric expressions in the program.	

3.4 Comments

Clear code reduces the need for comments. Self-explanatory code serves readability better than cheap and redundant comments. Unmaintained or obsolete comments abandoned throughout the source text may seriously hinder the readability of the code. Yet, up-to-date and concise comments inserted in the source code may ease review and maintenance so long as they provide additional and consistent information as to the actual intent of the code.

The practices are defined in the table below.

Id	Practice	Supporting example
032	Comments should be preferably placed in fixed locations throughout the source text so as to facilitate automated extraction.	
033	Educated use of the following classes of comments should be considered: • file headers • program unit specification headers • program unit body headers • data comments • statement comments • marker comments.	
033	Each source file shall contain a header stating:	

Id	Practice	Supporting example
	• ownership (author's name, contact references and position in the organisation) • context (project's reference, repository information) • revision history (summary of each change including the date, the name of the change author and the change outline). Note: It should be noted that automated version history is normally supported by modern configuration management systems.	
034	Each program unit specification shall contain a header describing: • purpose (what the unit does) • interface and effects (the complete interface of the unit, including specification of any global effects it can have) • dependencies (the statement of all dependencies of unit from other units in the program).	
035	Each program unit body shall contain a header describing: • operation (how the unit performs its function) • outline (summary of critical, complex or non self-explanatory algorithms) • implementation choices (record reasons for significant implementation decisions and / or list alternate implementation strategies).	
036	Comments on purpose and intended usage of all data types and objects in the program shall be	

Id	Practice	Supporting example
	supplied, unless their names are totally self-explanatory and their usage obvious.	
037	The same principle for the definition and use of all user-defined exceptions shall be applied.	
038	The semantic organisation and operation of data structures shall be explained unless totally obvious.	
039	The relationships that exist between data objects shall be documented. (Objects may be textually grouped by purpose to facilitate this.)	
040	Comments embedded among statements shall be minimised: comments should only be used to explain non-obvious parts of the code (e.g.: important algorithmic decisions).	
041	Comments shall not be used to paraphrase the code nor to attempt to explain what goes on inside remote units (e.g.: subprogram calls) as this violates the principle of information hiding.	
042	Pagination markers to signpost the boundaries of program units shall be used.	
043	Markers to tag the individual components of long (e.g.: nested) statements shall be used.	Example 8
044	Comments shall be used to highlight the presence of non-portable features in any unit of the program.	

CHAPTER 4

PROGRAM STRUCTURE

4.1 INTRODUCTION

Proper structure improves program clarity. Ada supplies various program structuring facilities which are designed to improved program clarity. Package concept and private types support abstraction and encapsulation. Strong typing and separation of program unit specifications from their bodies enforce information hiding.

An appropriate program structure minimises the amount of re-compilation needed after code changes. The specification defines the interface between the unit and all of its users. Recompilation of a specification may make recompilation of its users necessary to check for compliance. Experience says that bodies of units are updated far more frequently than their specification; this suggests that specification and bodies are best kept separate from one another. Furthermore, keeping bodies separate from their specifications may allow more bodies (i.e.: more implementations) to be defined for a single specification.

Ada 95 introduces further program structuring facilities to support more powerful hierarchies of library units than Ada 83 (notably, child libraries). Such novel facilities, however, are not discussed in the present edition of this standard. The interested reader is referred to [Ref. 2].

4.2 PROGRAM STRUCTURE

The practices are defined in the table below.

Id	Practice	Supporting Example
045	The specification of each library unit package shall be placed in a separate file from its body.	
046	A consistent file naming convention shall be adopted to highlight and reflect one such	For example the suffix .ads

Id	Practice	Supporting Example
	separation.	denotes a file containing a library unit package specification whereas the suffix .adb denotes its body)
047	The definition of library unit subprograms shall be strictly minimised.	
048	The only allowed library unit subprograms shall be those intended for use as main programs.	
049	Subprograms shall be used to enhance the abstraction of the program.	
050	Each subprogram shall be designed to perform one single action.	
051	Functions shall be used for side-effect-free subprograms whose primary purpose is to provide a single return value.	
052	Packages shall be regarded as the primary means to achieve: • information hiding • modelling of abstract data types and application-specific entities • grouping of related types, data and subprograms declarations • encapsulation of implementation dependencies • segregation of low-level implementation	

Id	Practice	Supporting Example
	decisions and program details.	
053	Packages shall be created to serve a single distinct purpose.	
054	Package specifications shall not contain variable declarations.	
055	The depth of nesting (iterations, branches, units) shall not exceed a maximum level expressed as a project-specific required practice.	

4.3 VISIBILITY

The ability to enforce information hiding and visibility restrictions is one of the most important strengths of the Ada language. Such a feature is useful and desirable and should not be subverted by unlimited reliance on the **use** clause.

The practices are defined in the table below.

Id	Practice	Supporting example
056	Package specifications shall only contain what the user actually needs to use (that is, the program shall avoid providing unnecessary visibility).	
057	The number of parameters in subprograms shall be minimised.	
058	The number of context (i.e.: **with**) clauses required for a package specification shall be minimised.	
059	The use of the **use** clause shall be minimised.	
060	The visibility of outer-scope data as a means to reduce the number of parameters to a	

Id	Practice	Supporting example
	subprogram shall be avoided.	
061	The effect of all used **use** clauses shall be localised. Note: The **renames** clause is a practical means to introduce short names, which reduce the complexity of fully qualified names. Use of the **renames** clause also achieves visibility within a scope without the need for the **use** clause. Renaming may also be used for other useful purposes e.g. limit the effect of overloading within a scope; make operations available through a primitive notation. Too liberal use of renaming, though, detracts from the readability of the program.	This is achieved, for example, by placement of the use clause in the body of the using unit or by encapsulation in a block.
062	The scope of renaming declarations shall be minimised. Note: Ada allows the use of the same name for different objects (e.g.: subprograms; operators) within the same scope. Any two such objects are called overloaded and the relevant practice is called overloading. This practice is legal so long as the interpretation of the context of use allows for the determination of exactly one meaning for the overloaded entity. Use of overloading requires a blend of programming skills and judicious consideration of the user needs, which makes it a rather daring practice.	
063	The use of overloaded subprograms shall be limited to the implementation of subprograms that are to perform similar or equivalent actions on arguments of different types.	
064	The use of overloaded operators shall preserve the conventional meaning of the operator in the	Example 9.

Id	Practice	Supporting example
	relevant algebra (e.g.: "+" must still mean adding, even under overloading).	

4.4 CONTROL FLOW

The primary control flow structures provided by the Ada language are the selection and iteration constructs. Selection is performed by the if and case statements. Iteration is performed by the basic loop ... end loop construct combined with the modifiers for or while.

The practices are defined in the table below.

Id	Practice	Supporting example
065	**`case`** statements shall only be used when the selection is made on the basis of one single value and the alternate actions are mutually exclusive. Under such conditions, **`case`** statements shall always be preferred to **`if/elsif`** constructs.	
066	Iterations shall be based on **`for`** constructs at all time when the maximum number of iterations is statically known. Note : **`for`** constructs are also more robust in that the provided iterator is of read-only type and its scope is limited to the construct itself.	
067	Iterations shall be based on **`while`** loops when the number of iterations cannot be calculated before entering the loop.	
068	Iterations shall not include invariant (i.e. iteration-independent) execution components.	
069	Termination of an iteration at an arbitrary point shall be achieved by use of the **`exit`** statement.	

Id	Practice	Supporting example
070	The number of ways to **`exit`** a loop shall be minimised.	
071	The use of: **`exit when`** `<condition>` shall be preferred to the use of: **`if`** `<condition>` **`then exit`**.	
072	The use of the **`goto`** statement shall be avoided in that it allows uncontrolled transfer of control across program scopes.	

CHAPTER 5
PROGRAMMING PRACTICES

5.1 OPTIONAL PARTS OF THE SYNTAX

The practices are defined in the table below.

Id	Practice	Supporting example
073	Nested loops shall be named.	
074	Loops which contain an **`exit`** terminator shall be named and the name used on the exit statement itself.	
075	The loop name shall be used on all **`exit`** statements from nested loops.	
076	The defining program unit name shall be included at the end of a package specification and body.	
077	The defining identifier shall be included at the end of a task specification and body.	
078	The entry identifier shall be included at the end of the relevant **`accept`** statement.	
079	The designator shall be included at the end of a subprogram body.	

5.2 PARAMETER LISTS

The parameter list to a subprogram or entry is the interface to that unit. It therefore needs to respect all rules aimed at improving readability and clarity of interface structures.

The practices are defined in the table below.

Id	Practice	Supporting example
080	Descriptive names shall be used to name formal parameters so as to reduce the need for explanatory comments.	
081	Named parameter association shall be preferred to positional association for all parameter lists greater than a project-specific limit (e.g.: 3).	Example 10.
082	The mode indication of all subprogram and entry parameters shall always be explicitly specified.	
083	The most restrictive parameter mode, as allowed by the application needs, shall be used.	
084	The number of arguments to subprograms and entries shall be limited. An excess of arguments to a subprogram or entry denotes the attempt to group several aims within that unit, which is a bad programming practice and detracts from the readability and reusability of the unit.	

5.3 TYPES

The use of strong typing enhances the reliability of software. The definition of the legal values and operations for a given type allows the compiler to perform static checks on their usage and identify potential usage errors. The provision of such definitions also allows the compiler to generate code which performs checks for violation of type constraints at execution time.

The set of legal values for an application should be regarded and enforced as a property of the program and should not depend on hidden properties of the language implementation. This is achieved by educated use of types, derived types and subtypes:

- predefined and new types primarily serve as building blocks for the creation of more specialised type derivatives
- derived types represent new types to all effect and are, therefore, different from their base type
- subtypes limit the range of possible values for the base type but do not define new types.

The most part of type checking in Ada is performed at compile time. The Ada user should therefore be aware that there is no direct relationship between the use of strong typing and the risk of execution penalties.

The practices are defined in the table below.

Id	Practice	Supporting example
085	The range of scalar types shall be as limited as possible (i.e.: shall eliminate as many meaningless values as possible).	
086	Enumeration types shall always be used instead of numeric codes. Where direct mapping to external values needs to be maintained, representation clauses shall be used to achieve it.	
087	The use of anonymous array types shall be avoided. Proper types shall be defined instead and used accordingly.	

5.4 DATA STRUCTURES

Ada has powerful data structuring capabilities. Grouping of logically related data in a building-block fashion (that is, creating data structures from smaller components) facilitates maintenance and enables reuse.

The practices are defined in the table below.

Id	Practice	Supporting example

Id	Practice	Supporting example
088	Heterogeneous but related data shall be grouped using records.	Example 11
089	Discriminated records capture the intent of arrays whose bounds may vary at run-time. Arrays with such characteristics shall be modelled using discriminated records rather than constrained arrays.	
090	Discriminants shall be based on as constrained a subtype as possible. This reduces the amount of useless (i.e.: wasted) space allocated to the object.	Example 12
091	The use of large flat record structures shall be avoided; wherever possible, related components shall be grouped into smaller subrecords. Complex data structures are better understandable if composed of meaningfully related building blocks that have been factored out.	Example 13
092	The use of dynamically allocated objects shall be carefully considered, preferably avoided, and in all cases justified. Note: Dynamic allocation (performed by execution of the allocator **`new`**) and deallocation of objects (performed by execution of **`Unchecked_Deallocation`** or implicitly caused by the object's access type going out of scope) involve heap management at run-time. Heap operations are intrinsically unbounded and potentially unpredictable; they are, therefore, antagonistic to the construction of high-integrity, predictable systems.	
093	If the use of dynamically allocated data structures is required, specific measures shall be enforced to ensure, at least, that:	

Id	Practice	Supporting example
	• no dangling references are left to deallocated objects • pointers to undeallocated objects are not dropped • all deallocations are performed explicitly • all access variables are initialised explicitly • handlers are provided for **`Storage_Error`** exceptions • very precise information is provided as to whether and how the run-time system reclaims heap storage.	
094	The use of uninitialised objects is a breach to program's integrity. All variable objects shall, therefore, be initialised prior to use. This is best achieved by assigning objects an initial value at the time of declaration.	
095	Entities in an Ada program need to be elaborated before being ready for use. Care shall be taken to ensure the elaboration of an entity before using it.	Example 14

5.5 EXPRESSIONS

Properly coded expressions enhance the readability and understandability of a program.

The practices are defined in the table below.

Id	Practice	Supporting example
096	The use of constructs and expressions whose meaning or effect depends on the order of evaluation shall be avoided.	

Id	Practice	Supporting example
097	Parentheses shall be used to specify and clarify the desired order of evaluation in expressions.	Example 15
098	Relational expressions shall be expressed, as far as possible, in a positive form.	Example 16
099	Conditional constructs shall be ordered such that the most frequently used branch is encountered first.	
100	The use of short-circuit control forms helps one prevent data-dependent errors or exceptions that may result from expression evaluation. Short-circuit forms of the conditional construct shall be used to specify the order of condition evaluation when the failure of one condition means that the other becomes unfeasible (e.g.: raises an exception).	Example 17
101	The hardware representation of real numbers is finite and imprecise. Round-off errors may result from variation in the construction path or history of any two real numbers. Relational expressions shall, therefore, use **<=** or **>=** instead of **=**.	

5.6 STATEMENTS

Careless or convoluted use of statements can make a program difficult to read and maintain even if its global structure is well organised.

The practices are defined in the table below.

Id	Practices	Supporting example
101	The depth of nested expressions shall be	

Id	Practices	Supporting example
	minimised.	
102	The depth of nested control structures shall be minimised.	
103	Slices should be used to copy part of an array.	
104	Aggregates shall be used instead of a sequence of assignments to assign values to all components of a record.	Example 18.
105	The use of **others** choice is **case** statements shall be minimised.	
106	Ranges of enumeration literals shall not be used in **case** statements.	
107	The number of **return** statements from a subprogram shall be minimised. Good coding practice strives to keep this number to 1 at all times.	
108	**Return** statements shall be highlighted (e.g.: with comments) so as to make them clearly distinguishable from other code.	
109	**Unchecked_Conversion** is a bit-for-bit copy operation (implemented as a predefined generic library function) that does not attach meaning to the bit-wise value of the object at both the source and the destination type. **Unchecked_Conversion** can be extremely useful for hardware-orientated data manipulation (e.g.: data bus handling). Care shall, however, be taken to avoid that the source pattern be meaningless in the context of the destination type or violate type constraints in subsequent operations. Note: It is good practice to attach explanatory comments to the declaration and use of every **Unchecked_Conversion** operation. Such	

Id	Practices	Supporting example
	comments should clarify: all of the pre-requisites and assumptions which hold on the expected contents of the source data (e.g.: range, meaning and authentic interpretation); the rationale for the choice of the destination type; and the precautionary measures taken to avoid the occurrence of constraint violations.	

5.7 GENERICS

Ada generic units provide a means to define general-purpose algorithms that are designed to be reused across programs (the relevant form of reuse is called horizontal reuse and is aimed to span across diverse application domains).

The use of generic units, however, is not the only way to promote reuse in Ada. Factorisation of data structures and related operations is a more natural and effective means to support reuse of building blocks across one program or programs within the same application domain.

Efficient coding of generic unit takes careful consideration of the desired entity or operation and considerable algorithmic skills.

Differing solutions may be adopted by the compiler to support instantiation of generic units (e.g.: the "macro-expansion" method produces an instantiation by physical copy of the unit's code as opposed to the "shared-bodies" method, which maintains a unique copy of the unit). Differing implementation methods may have differing effects at execution time (e.g.: memory consumption versus execution speed) which need to be carefully considered by the user.

The practices are defined in the table below.

Id	Practice	Supporting example
110	Generic units shall be organised as collection of library units.	

111	Information shall be obtained from the compiler vendor as to the adopted method to support instantiation of generic units.	
112	Careful consideration shall be given to the opportunity and resource implications at run-time of requiring instantiation of generic units.	

5.8 EXCEPTIONS

Ada exceptions are language features designed to help one specify the intended program behaviour in the face of errors or unexpected events. The use of exceptions, however, is provably not sufficient to provide for the comprehensive implementation of software fault tolerance.

The practices are defined in the table below.

Id	Practices	Supporting example
113	Exceptions shall not be used as a general-purpose control flow modifier.	
114	Care shall be taken to avoid deliberate or implicit raising of exceptions. The use of defensive programming is a suitable means to do so.	Examples of defensive programming techniques are: • assertive verification of an object's correct value prior to use • localisation of potentially exception-raising operations

Id	Practices	Supporting example
		within enclosing blocks provided with exception handlers.
115	Handlers shall be provided for all the exceptions that cannot be avoided and all that are explicitly allowed. Note: When execution performance is an issue, it may be more efficient to perform potentially exception-raising operations within the scope of an exception handler than to exclusively rely on defensive programming.	
116	Care shall be taken to avoid undesired propagation of exceptions. Note: Predefined and implementation-defined exceptions are natural candidates for propagation to higher levels of abstractions; it may, in fact, prove difficult to determine exactly which statement and which operation within that statement raised an exception. User-defined exceptions are more easily associated with useful recovery actions by localising handlers within small blocks of code (e.g.: a subprogram body).	
117	The use of **`others`** exceptions shall be avoided or just limited to development for the purpose of capturing, qualifying and providing specific handlers for potential exceptions.	
118	Exception checks is a powerful debugging aid during development. Exception checks shall not be suppressed	

Id	Practices	Supporting example
	during development. Note: Removal of exception checks during operation is a very questionable practice. With exception checks disabled, program execution which results in a condition where an exception would otherwise occur becomes erroneous and its results are unpredictable.	
119	Disabling of exception checks during operation shall be duly justified and its possible consequences carefully assessed.	

5.9 CONCURRENCY

A large number of application problems naturally map to a concurrent programming solution. In a single-processor environment, (apparent) concurrency is achieved by interleaved execution of concurrent activities. In a multi-processor environment, (real) concurrency is achieved by overlapped execution of concurrent activities.

Ada provides features that allow the modelling of both co-operating and independent concurrent activities. Ada tasks model concurrent activities and rendez-vous support synchronisation between co-operating concurrent tasks.

Such features, however, need to be used with extreme care (that is, in accordance with a well-defined programming model) in order to avoid the occurrence of unpredictable program behaviour at execution time (e.g.: deadlocks, unbounded blocking, priority inversion).

A number of predictable instances of concurrent programming models exist in the literature and have successfully been applied in space-related application programs. [Ref. 6], for example, describes one such model which suits the concurrent implementation of real-time embedded applications. This proves that the Ada concurrency constructs can be effectively used in a manner which guarantees the application the desired extent of predictability.

The effort to make educated direct use of the Ada concurrency features is a far more advisable practice than the attempt to negotiate with the language implementation for the construction of alternate forms of concurrency. The former, in fact, produces portable code and enjoys the support of the associated technology (compiler and run-time checks, debugging aid) to a much greater extent than the latter.

5.9.1 Tasking

The practices are defined in the table below.

Id	Practice	Supporting example
120	The use of tasks to model naturally concurrent activities within the problem domain should be considered.	
121	The use of anonymous tasks avoids proliferation of task types that are used only once: unique instances of concurrent tasks shall be declared using single-task declaration.	
122	The use of dynamic task creation shall be minimised and preferably avoided in that it may incur a potentially high start-up overhead.	
123	The use of tasks in time-critical applications shall adhere to a well-defined programming model that allows for static and dynamic verification of the desired properties of execution.	
124	Care shall be taken to assess the effect and the guaranteed accuracy of voluntary suspensions achieved by use of the **`delay`** statement.	

5.9.2 Priorities

Task scheduling in Ada assumes a priority-based pre-emptive policy. The assignment of priorities to tasks, however, needs to be educated by accurate knowledge of the actually implemented scheduling policy and careful assessment of the effects of pre-emption and synchronisation incurred at program execution.

The practices are defined in the table below.

Id	Practice	Supporting example
125	Practical recommendations for the use of priorities shall be provided as part of any suitable programming model which addresses the use of tasking.	
126	No unproven reliance shall be made on the support to **`pragma Priority`** and priority scheduling provided by the compiler.	
127	The feasibility and suitability of the recommended practices for the use of priorities adopted as part of the tasking-oriented programming model shall be explicitly verified against the support provided by the compiler.	

5.9.3 Communication

Ada provides a rich set of constructs to model various forms of synchronisation between co-operating tasks. Not all such forms of synchronisation equally suit differing application domains: time-critical applications, for example, will probably need to define tight restrictions on the allowable type of synchronisation.

The practices are defined in the table below.

Id	Practice	Supporting example
128	Care shall be taken to establish the set of synchronisation constructs appropriate for the specific application.	
129	Recommendations for the use of synchronisation constructs shall be provided as part of any suitable programming model which allows the use of co-operating tasks.	
130	Entries of a task shall not directly or indirectly call entries of the original calling task because this	

Id	Practice	Supporting example
	determines deadlock situations.	
131	The work performed during rendez-vous shall be minimised as it blocks both the caller and the callee.	
132	Careful consideration shall be given to the opportunity of catering for exception handling during rendez-vous.	
133	The work performed in the selective **`accept`** loop of a server task shall be minimised as it increases the time it takes to return to the next accept, in addition to being prone to potential suspension of the server.	
134	Shared variables shall not be used as a task synchronisation device: guards shall only reference local variables.	
135	The number of **`accept`** and **`select`** statements within a task shall be minimised.	
136	The number of **`accept`** statements per **`entry`** shall be minimised. (Ideally, there shall be a one-to-one correspondence between an **`entry`** and its **`accept`** statement.)	

5.9.4 Termination

The definition of the conditions under which tasks may need to terminate is an important design decision.

Very few guidelines regarding task termination can be given which have a sufficiently general applicability.

The practices are defined in the table below.

Id	Practice	Supporting example
137	The use of an exception handler for a rendez-vous	

	within the main loop of a task shall be considered as a desirable means to avoid unwanted task termination.	
138	The use of an **`others`** exception hander at the end of a task body shall be considered as a desirable means to avoid abnormal task termination.	
139	The use of **`select`** statements with a **`terminate`** alternative shall be considered as a desirable means to obtain controlled task termination.	
140	The use of the **`abort`** statement shall be avoided.	

CHAPTER 6
DEPENDENCIES

6.1 INTRODUCTION

The achievement of program portability takes a very detailed understanding of the Ada model and its implementation. Furthermore, the recent introduction of the revised Ada standard has caused the issue of portability of Ada programs to involve the two complementary dimensions of portability of the same program across differing computer systems and of Ada 83 programs to Ada 95 implementations. The present standard in its current edition does not address the latter and limits itself to a general discussion of the former.

6.2 FUNDAMENTALS

Four general principles can be laid down which help one lessen the effect of dependency on implementation-sensitive Ada constructs. Such principles are the following:

- recognise the Ada constructs that may adversely affect portability across implementations or computing platforms
- rely as much as possible and practical on Ada constructs whose support characteristics are shared by all the relevant implementations
- localise and encapsulate (by use of the Ada's structuring facilities) the non-portable features of a program whose use is essential
- highlight (by use of explanatory comments) the use of constructs that may cause portability problems.

The practices are defined in the table below.

Id	Practice	Supporting example
141	Informed assumptions shall be made about the support provided for:	

Id	Practice	Supporting example
	• number of bits available for type **`Integer`** • number of decimal digits of precision available for floating-point types • number of bits available for fixed-point types • number of characters per line of source text.	
142	Highlighting comments shall be used to mark the use of non-portable features.	
143	Packages shall be specifically created to isolate and encapsulate hardware and implementation dependencies.	
144	Packages encapsulating dependencies shall be designed so as to resolve dependencies within target-specific bodies thereby allowing specifications not to change when porting.	
145	Code whose correct execution depends on implementation-specific properties (e.g.: parameter passing mechanisms) shall be avoided.	

6.3 NUMERIC TYPES AND EXPRESSIONS

Ada is a language expressly designed for use in embedded systems and mathematical applications, where precision is important. Ada, however, was also designed to maximise the portability of applications.

The practices are defined in the table below to avoid these potentially conflicting aims.

Id	Practice	Supporting example
146	The strategy of use of the numeric types and expressions in Ada shall be determined in accordance with the driving requirements on the	

Id	Practice	Supporting example
	application. Note: Effective use of floating-point or fixed-point operations requires the determination of the accuracy provided by the specific implementation of the language.	
147	Reliance on general-purpose predefined numeric types should be avoided. The language typing facility should be used to create the numeric types appropriate to the problem. Note: Floating-point operations are often directly supported by the hardware of modern platforms and the resulting precision is relatively straightforward to determine for any given implementation. Within the capability of the hardware, clause **`digits`** determines the accuracy required of the associated floating-point type. The Ada fixed point, instead, requires more extensive compiler support and, hence, it is more prone to diversities between implementations. The required precision of the fixed-point type is set by use of clause **`delta`**. Simple means exist, which help one detect peculiarities of the fixed-point implementation in any given language implementation (c.f., for example, [Ref. 7].)	
148	Reliance on the default accuracy of fixed-point or floating-point types shall be avoided. The language facility (**`digits, delta`**) shall be used to specify the required precision of the type. Note: Further control on the precision of fixed-point and floating-point types can be obtained by verification of the type's attributes.	
149	Avoid modifying the default value of the fixed-point and floating-point type's attributes. Some of these attributes, in fact, play a key role in the	

Id	Practice	Supporting example
	accuracy of the operations on the type (e.g.: **`small`**).	

6.4 REPRESENTATION CLAUSES

Ada provides the means for extensive control over and interactions with the underlying hardware architecture. The representation clauses are expressly designed for that purpose. Appropriate use of representation clauses instructs the compiler on the required size, layout and mapping of entities (types, objects, subprograms) of the program.

Embedded applications have vastly different needs in this respect from those of conventional host-based applications. The present standard guidelines cannot be intelligently specific for either of them.

The practices are defined in the table below.

Id	Practice	Supporting example
150	Project-specific complements to this standard, for the practices of use of representation clauses appropriate to the application domain, shall be supplied.	
151	The use of a representation clause (representation attribute **`storage_size`**) to specify the number of storage units for an object shall be avoided, unless portability is not a concern.	
152	Packages to encapsulate the entities of the program which are to be designed using representation clauses shall be used.	
153	The use of highlighting comments to document the intent, rationale and justification of every instance of use of a representation clause shall be applied.	Example 19

6.5 IMPLEMENTATION-DEPENDENT FEATURES

The use of vendor-specific packages hinders the portability of the code. The practices are defined in the table below.

Id	Practice	Supporting example
154	The language standard shall be applied as much as possible.	
155	Highlighting comments shall be used to mark those constructs which are non-standard.	

6.6 PRAGMAS

Pragma clauses are a distinct means to convey information to the compiler. Pragma clauses can be used, for example, to: request that certain object attributes be set to some initial value (e.g. **`: pragma priority`**); request the compiler to flag the possible occurrence of elaboration-order problems (**`pragma elaborate`**); request the compiler to omit the generation of certain run-time checks (**`pragma suppress`**).

The presence or absence of certain pragma clauses may have an important effect on the run-time behaviour of the program.

This standard takes the view that the use of pragma clauses is a project-specifc practice. Conformance with ad-hoc project-specific guidelines in this respect, therefore, takes precedence over any generic standard.

The practices are defined in the table below.

Id	Practice	Supporting example
156	Pragma clauses shall be used in strict conformance with project-specific guidelines.	
157	Reliance shall be placed on project-specific guidelines for the use of the language-defined pragma clauses. Note: Specific language implementations are allowed	

Id	Practice	Supporting example
	to support additional pragma clauses.	
158	Reliance shall be placed on project-specific guidelines for the use of the implementation-defined pragma clauses.	
159	Highlighting comments shall be used to mark the use and intent of each pragma in the program.	
160	Highlighting comments shall be used to flag every use of implementation-defined pragma clauses in the program.	

This page is intentionally left blank.

APPENDIX A CODING PRACTICE EXAMPLES

A.1 EXAMPLES OF CODING PRACTICES

This coding standard uses the following typographic conventions:

- **boldfaced** font is used for Ada reserved words when used in plain text
- `lowercase 10pt` font characters are used to denote Ada reserved words in code fragments
- `10pt` font words beginning with capital letter and separated by underscore are used to denote Ada identifiers in code fragments.

Example 1 Indentation of Nested Control Structures :

```
.
.   if  <condition> then
.       <statements>
.    elsif <condition> then
.       <statements>
.    else
.       <statements>
.    end if;
.
.    case <expression> is
.       when <choice> =>
.            <statements>
.       when <choice> =>
.            <statements>
.       when others =>
.            <statements>
.    end case;
```

Example 2 : Align operators vertically

```
.    if Slot_A > Slot_B then
.       Temporary := Slot_A;
.       Slot_A    := Slot_B;
.       Slot_B    := Temporary;
```

```
.    end if;
```

Example 3 : Alignment of parameter modes

```
.
.   procedure Display_Menu ( Title : in String;
.                            Options : in Menu;
.                            Choice : out User_Input);
.
```

Example 4 : Use of blank lines for grouping

```
.   type Employee_Record is
.      record
.         Legal_Name : Name;
.         Date_Of_Birth : Date;
.         Date_Of_Hire : Date;
.         Position : Grade;
.    end record;
.
.   type Day is
.      ( Monday, Tuesday, Wednesday, Thursday, Friday,
.        Saturday, Sunday );
.
.   subtype Weekday is Day range Monday    .. Friday;
.   subtype Weekend is Day range Saturday .. Sunday;
```

Example 5 : Use of underlines

```
.
.   type Allowed_Range is new integer range 1_000 .. 1_000_000;
```

Example 6 : Use of based literals

```
.   Default_Configuration : constant := 2#1001_1000# -- represent default
.                                                    -- configuration of
.                                                    -- an 8-bit h/w
.                                                    -- register
```

Example 7 : Declaration and use of named numbers

```
.   One_Third : constant := 1.0 / 3.0; -- a named number which evaluates
.                                      -- to universal_real
```

Example 8 : Use of markers for tagging

```
.
.    if Option_A then
.       ...
.    elsif Option_B then
.       ...
.    else -- no valid option chosen
.       ...
.       if Can_Do then
.          ...
.       end if; -- Can_Do
.       ...
.    end if; -- Option_A
.
.   ----------------------------------
.   package body Components is
.      ...
.   -------------------------------
.      procedure Search ( ... ) is
.         ...
.      begin
.         ...
.      end Search;
.    ------------------------------
.
.      ...
.   begin -- Components
.      ...
.   end Components;
.   ----------------------------------
```

Example 9 : Use of overloaded operators

```
.   type Address is private;
.   type Offset is range 0 .. 1_024;
.
.   function "+" (Left : in Address;
.                 Right : in Offset) return Address;
.   function "+" (Left : in Offset;
.                 Right : in Address) return Address;
```

Example 10 : Named parameter association

```
.
.   Encode_TM_Packet ( Source        => Thermal_Subsystem,
.                      Content       => Temperature,
.                      Value         => Read_Sensor ( Thermal_Subsystem ),
.                      Time          => Current_Time,
.                      Sequence      => Packet_ID,
.                      Configuration => Main );
```

Example 11 : Grouping heterogeneous data

```
.
.    type Payment_Method is (Cache, Check, Card);
.
.    type Customer is
.       record
.          Name : Family_Name;
.          Pay : Payment_Method;
.          Amount : Currency;
.       end record;
.
.    type Inventory is array ( 1 .. Inventory_Size ) of Customer;
.
```

Example 12 : Discriminants

```
.
.   Max_Length : constant := 42; -- list contains at most 42 items
.                                -- but objects of type Item_Holder
.                                -- may have smaller lists
.
.    type    Item_List is array ( Integer range <> ) of Integer;
.    subtype Max_Items is Integer range 1 .. Max_Length;
.
.    type Item_Holder ( Current_Length : Max_Items := 1 ) is
.       record
.          Items : Item_List ( 1 .. Current_Length );
.       end record;
.
```

Example 13 : Composing complex data blocks

```
.
.   type Coordinate is
.     record
.        X : Axis_Length;
.        Y : Axis_Length;
```

```
.      end record;
.
.    type Bounding_Box is
.       record
.          Top_Left : Coordinate;
.          Bottom_Right : Coordinate;
.       end record;
.
```

Example 14 : Elaboration of an entity

```
.    ------------------------------------------------
.    package Controller is
.        ...
.    function Acquire_Position return Position;
.        ...
.    end Controller;
.    ------------------------------------------------
.    package body Controller is
.       ...
.       Initial_Position : Position := Acquire_Position; -- This raises
.          -- Program_Error
.       ...
.    ------------------------------------------------
.    function Acquire_Position return Position is
.       begin
.       ...
.    end Acquire_Position;
.    ------------------------------------------------
.    begin -- Controller
.       Initial_Position : Position := Acquire_Position; -- This is correct
.                                                        -- as the function
.                                                        -- has now been
.                                                        -- elaborated
.       ...
.    end Controller;
```

Example 15 : Use of parentheses to aid evaluation of expressions

```
.
.    -- the expression:
.    5 + ( ( Y ** 3 ) mod 10 )
.
.    -- is clearer, and equivalent, to:
.    5 + Y ** 3 mod 10
```

Example 16 : Positive expression of rational terms

```
.
.    -- the following conditional construct:
.    if Item_Not_Found then
.
.    -- shall be preferred to:
.    if not Item_Found then
```

Example 17 : Use of short-cut symbol

```
.
.    -- use of the following:
.    if Y /= 0 and then ( X / Y ) /= 10 then
.
.    -- avoids the occurrence of Constraint_Error
.    -- which may result from use of:
.    if ( Y /= 0 and ( X / Y ) /= 10 ) then
```

Example 18 : Use of aggregates in assignments

```
.
.   Employee : Employee_Record := ( Legal_Name    => “White, J.B.”,
.                                   Date_of_Birth => “34-06-12”,
.                                   Date_of_Hire  => “95-02-01” );
```

Example 19 : Commenting with representation clause

```
.
.   Word_Size : constant := 32;
.
.   type Integer_32 is range – 2 ** (Word_Size – 1) .. 2 ** (Word_Size – 1) – 1;
.   for Integer_32’size use 32
.      – so that Integer_32 can map to a whole 32-bit register
.
.   type Registers is … -- implementation-specific! (details omitted)
.
.   procedure Write_Register ( Value : in Integer_32
.                              Destination : in Registers );
```

APPENDIX B REFERENCES

1. Ada Reference Manual. International Standard ANSI/ISO/IEC-8652:1987.

2. Ada Reference Manual. International Standard ANSI/ISO/IEC-8652:1995.

3. ISO JTC1/SC22 Working Group 9 (Ada). PDTR 15942 Project Nr: 15942, Title: "Guidance for the use of the Ada Programming Language in High Integrity Systems", Editor: Dr Brian Wichmann.

4. Ada 95 Rationale. Intermetrics Inc. Cambridge, MA. January 1995.

5. Ada 95 Quality and Style: Guidelines for Professional Programmers. Department of Defense Ada Joint Program Office. SPC-94093-CMC Version 01.00.10. October 1995.

6. T. Vardanega. Tool Support for the Construction of Statically Analysable Hard Real-Time Ada Systems. Technical Report. Available under heading “User Documentation” from the author at ESTEC.

7. B. Wichmann. Testing Ada fixed point operations. NPL Report CISE 11/97. February 1997.