

# *M68K Ada Technical Summary*

**For mission-critical applications on Motorola  
M68000 family computers**





---

# *M68K Ada Technical Summary*

**For mission-critical applications on Motorola  
M68000 family computers**

**Order Number: M68K-ADA-TS-040402**

**XGC Technology**

**London  
UK**

**<[www.xgc.com](http://www.xgc.com)>**

---

# **M68K Ada Technical Summary: For mission-critical applications on Motorola M68000 family computers**

by Chris Nettleton and Ellis Thomas

Publication date April 2, 2004

© 1998, 1999, 2000, 2001, 2002, 2003, 2004 XGC Technology

## **Abstract**

This report presents technical and commercial information about Version 1.7 of the M68K Ada compilation system.

## **Acknowledgments**

XGC Technology acknowledges contributions from the following organizations:

- The European Space Agency, contracts 11935 and 11374
- The UK Ministry of Defence, HOLD III contract
- TRW Aerospace, HOLD III contract
- New York University and ACT, Inc., for the GNAT front end
- The Free Software Foundation, for the C compiler, assembler and linker.

## **Notice**

The information in this document is subject to change without notice and should not be construed as a commitment by XGC Technology. XGC Technology assumes no responsibility for any errors that may appear in this document.

---

---

# *Contents*

---

## *Preface* **xiii**

### **Chapter 1**

#### *Introduction* **1**

- 1.1 Performance **2**
- 1.2 Restrictions **2**
- 1.3 User Documentation **2**
- 1.4 Media **3**
- 1.5 Warranty **3**

### **Chapter 2**

#### *Host and Target* **5**

- 2.1 Cross-Development System **5**
- 2.2 Host Configurations **6**
- 2.3 Host Operating System **6**
- 2.4 Target Configurations **6**
- 2.5 Target Operating System **7**
- 2.6 Programming Support Environment **7**
- 2.7 Host-Target Communication **7**

Chapter 3

*Language-Related Issues* 9

- 3.1 Overview 9
- 3.2 Section 2: Lexical Elements 10
- 3.3 Section 3: Declarations and Types 11
  - 3.3.1 Uninitialized Variables 11
  - 3.3.2 Enumeration Types 11
  - 3.3.3 Integer Types 12
  - 3.3.4 Floating Point Types 13
  - 3.3.5 Fixed Point Types 14
- 3.4 Section 4: Names and Expressions 14
- 3.5 Section 5: Statements 15
- 3.6 Section 6: Subprograms 15
- 3.7 Section 7: Packages 15
- 3.8 Section 8: Visibility Rules 15
- 3.9 Section 9: Tasks and Synchronization 15
  - 3.9.1 type Duration 16
  - 3.9.2 Shared Variables 17
- 3.10 Section 10: Program Structure and Compilation Issues 17
- 3.11 Section 11: Exceptions 18
- 3.12 Section 12: Generic Units 18
- 3.13 Section 13: Representation Issues 18
  - 3.13.1 Definitions from the predefined package System 19
  - 3.13.2 The type Address 20
- 3.14 Input-Output 20
- 3.15 Annex A: Predefined Language Environment 20
- 3.16 Annex B: Interface to Other Languages 22
- 3.17 Annex C: Systems Programming 22
- 3.18 Annex D: Real-Time Systems 23
- 3.19 Annex E: Distributed Systems 24
- 3.20 Annex F: Information Systems 24
- 3.21 Annex G: Numerics 24
- 3.22 Annex H: Safety and Security 24
- 3.23 Annex J: Obsolescent Features 25
- 3.24 Annex K: Language-Defined Attributes 25
- 3.25 Annex L: Language-Defined Pragmas 25

<b>Chapter 4</b>	<i>User Interface and Debugging Facilities</i>	<b>27</b>
	4.1 Compiler Invocation	27
	4.2 Compilation	28
	4.2.1 Format and Content of User Listings	28
	4.3 Errors and Warnings	30
	4.4 Other Software Supplied	31
	4.5 Debugging Facilities	33
<b>Chapter 5</b>	<i>Performance and Capacity</i>	<b>35</b>
	5.1 Host Performance and Capacity	35
	5.2 Target Code Performance	36
	5.2.1 Optimization and Code Quality	37
	5.2.2 Constraint Checks	38
	5.2.3 Space for Unused Variables	39
	5.2.4 Space for Unused Subprograms	39
	5.2.5 Evaluation of Static Expressions	39
	5.2.6 Elimination of Unreachable Code	39
	5.2.7 Common Sub-expressions	39
	5.2.8 Loop Invariants	39
	5.2.9 Bound Checks	40
	5.2.10 The pragma Inline	40
	5.2.11 Procedure Calling Overhead	40
	5.2.12 The Rendezvous	40
	5.2.13 Space Requirements	41
<b>Chapter 6</b>	<i>Cross-Compiler and Run-Time Interfacing</i>	<b>43</b>
	6.1 Cross-Compiler Issues	43
	6.1.1 Background	43
	6.2 Compiler Phase and Pass Structure	44
	6.3 Compiler Module Structure	45
	6.3.1 Intermediate Program Representations	45
	6.3.2 Final Program Representation	45
	6.3.3 Compiler Interfaces to Other Tools	46
	6.4 Compiler Construction Tools	46
	6.5 Installation	47
	6.6 Run-Time System Issues	47
	6.6.1 The Stack	47

	6.6.2	Subprogram Call and Parameter Handling	47
	6.6.3	Data Representation	48
	6.6.4	Implementation of Ada Tasking	49
	6.7	Exception Handling System	50
	6.8	I/O Interfaces	50
	6.9	Documentation	50
<b>Chapter 7</b>		<i>Re-targeting and Re-hosting</i>	<b>51</b>
	7.1	Retargeting	51
	7.2	Rehosting	52
	7.2.1	Availability of Source Code	52
	7.2.2	Source Language	52
	7.2.3	System Dependencies	52
<b>Chapter 8</b>		<i>Contractual Matters</i>	<b>53</b>
	8.1	The Compiler License	53
	8.2	The Run-Time License	54
	8.3	Support	54
<b>Chapter 9</b>		<i>Validation</i>	<b>57</b>
<b>Appendix A</b>		<i>Examples of Generated Code</i>	<b>59</b>
	A.1	The Sieve of Eratosthenes	59
	A.2	Ackermann's Function	62
<b>Appendix B</b>		<i>Restrictions and Profiles</i>	<b>65</b>
<b>Appendix C</b>		<i>The Predefined Library</i>	<b>71</b>



---

# Tables

3.1	Attributes of the Predefined Integer Types	<b>12</b>
3.2	Basic Attributes of Floating Point Types	<b>14</b>
3.3	Attributes of the Predefined Type Duration	<b>16</b>
3.4	Named Numbers from package System	<b>19</b>
5.1	Benchmark Results	<b>36</b>
5.2	Task-Related Metrics	<b>37</b>
9.1	The Validation Test Classes	<b>57</b>
B.1	Supported Profiles	<b>66</b>
B.2	Profiles and Restrictions	<b>66</b>
B.3	Profiles and Numerical Restrictions	<b>68</b>
C.1	Predefined Library Units	<b>71</b>



---

## Examples

- A.1 Source Code for Sieve **59**
- A.2 Generated Code for Sieve **60**
- A.3 Ada Source Code for Ackermann's Function **63**
- A.4 Generated Code for Ackermann's Function **63**



---

# *Preface*

---

This summary provides technical information about the M68K Ada cross compiler. It is intended for anyone evaluating cross compilers for development environments using workstations running the UNIX operating system, and microprocessor targets. The reader is expected to be familiar with the Ada 95 programming language.

The information in this summary is organized according to the *Ada-Europe Guidelines for Ada compiler specification and selection*. These guidelines pose questions about an Ada implementation that are designed to assist vendors and users of Ada compilers. Although written for Ada 83, these guidelines continue to be relevant for Ada 95, and for this summary, we include answers to any Ada 95-specific questions.

Questions from the guidelines are not restated; topics are discussed in a manner that makes it unnecessary to refer to the original questions. Supplementary information is provided as appropriate. An appendix shows listing from two small compilations to help answer many of the questions related to compilation listings and

error messages. The presentation is terse to provide as much information as possible in a compact form.

The *Ada-Europe Guidelines for Ada compiler specification and selection* were written in 1982 by J.C.D. Nissen, B.A. Wichmann, and other members of Ada-Europe, with partial support from the Commission of the European Communities. They are available from the National Physical Laboratory as NPL report DITC 10/82, ISSN 0262-5369. They were also reprinted in *Ada Letters*, Vol. III, No. 1 (July, August 1983), pp. 37-50. (*Ada Letters* is published every two months by SIGAda, the ACM Special Interest Group on Ada.)

**Version 1.7.** Version 1.7 adds exception handling but does not support exception propagation (down the dynamic stack). Handlers can only handle exceptions raised locally. Version 1.7 also adds a static subset of programming by extension. Dispatching is not supported and the attribute `'Class` is prohibited.

**Version 1.6.** Version 1.6 offers broader functionality with a smaller run-time system. The default profile is extended with allocators, catenation operators and the Ada 83 rendezvous. Functions that return unconstrained types are also permitted. So-called “emulations” are added for systems that execute from EPROM, or copy code from EPROM to RAM and execute it there.

**Version 1.5.** Version 1.5 includes further support for real-time systems. Several Ada child packages that were previously absent are now available in the default profile. We have added the pragma Profile, which offers a choice of five mission-critical profiles. Note that M68K Ada still prohibits non-static tasks, the rendezvous, allocators and exception handlers, and other Ada features that depend on these.

**Version 1.4.** For version 1.4 the standard target is the XGC simulator. This simulates the instructions of any member of the Motorola M68000 family.

**Version 1.3.** Version 1.3 introduces user-mode tasks, and limits potentially dangerous operations to supervisor mode. Tasks use traps to communicate with the run-time system, which may be loaded as a separate memory image.

---

**Version 1.2.** The main change since Version 1.1 is the addition of a limited form of Ada tasking that supports the *Ravenscar Profile*. The profile includes tasks and protected objects declared in library packages, and several features from Annexes C and D.

**Version 1.1.** Version 1.1 is developed for safety-critical engine-control applications as part of the HOLD III project. This work is funded by Lucas Aerospace and the Ministry of Defence. The target is the Motorola MC68020 with no floating point unit.





---

M68K Ada is a cross-development system providing a production-quality implementation of a restricted Ada 95 language (ANSI/ISO/IEC-8652:1995). Significant features of M68K Ada are as follows:

- Minimum program size 2K bytes
- Accurate delays with 100 microseconds<sup>1</sup> delay latency over whole range
- Low overhead 4K byte tasking system with 50 microseconds<sup>1</sup> task switch
- Full support for interrupts attached to protected subprograms
- Comprehensive printed and on-line user manuals
- Available off the shelf as a fully supported commercial product

---

<sup>1</sup>Simulated MC68040 at 25 MHz

- Evaluation copies available for down-load
  - Compatible with other XGC Ada compilation system
  - Built-in restrictions for mission-critical applications (see Appendix B, *Restrictions and Profiles* [65])
- 

### 1.1. Performance

M68K Ada includes a high-performance run-time system that optionally supports Ada tasking, interrupt handling and real-time scheduling. For more information on the real-time performance, see Chapter 5, *Performance and Capacity* [35].

---

### 1.2. Restrictions

Several sets of restrictions are supported. These are known as *profiles*, and may be employed to ensure an appropriate level of software integrity. For more information on restrictions and profiles see Appendix B, *Restrictions and Profiles* [65].

---

### 1.3. User Documentation

The documentation provided with M68K Ada includes the following:

- *Getting Started with M68K Ada*, which describes how to write and run a small application program.
- *XGC Ada Language Reference Manual Supplement*, which includes implementation-specific information required by the Ada standard.
- *XGC Ada User's Guide (three volumes)*, which describes how to use the XGC Ada toolset.

All documentation is shipped in source format (XML), HTML format, and Adobe® PDF format.

---

### *1.4. Media*

M68K Ada is shipped on CD-ROM, and includes both on-line and printed user manuals.

---

### *1.5. Warranty*

M68K Ada includes six months support to help users install and become familiar with the compiler and with using the Ada language on M68000 computers.



---

This chapter gives details of the following:

- the host configurations on which the compiler will run
- the target configurations on which compiled programs will run
- the means for transferring a compiled program from the host computer to the target computer.

---

### *2.1. Cross-Development System*

A cross development system is used where programs written on one machine are compiled to run on another. The machine used for software development is the *host* and the machine on which the programs run is the *target*.

Typically, this form of development is associated with embedded software for real-time applications. This approach enables the target

computer to be optimized for the embedded application and the development tools to exploit the effectiveness of the host computer.

---

## *2.2. Host Configurations*

The host computer should be a UNIX workstation or personal computer that meets the following minimum requirements:

- 50MHz, 32-bit CPU
- 1G byte hard disk drive
- 24M bytes RAM
- High-resolution monitor with X Windows and window manager
- Network interface supporting TCP/IP
- Serial interface for host-target link

By adding extra terminals, a system like this can support several users at the same time.

---

## *2.3. Host Operating System*

The standard host operating systems are as follows:

- Solaris® 2.6 or above, running on a Sun SPARC® computer.
- RedHat® Linux Version 7.3 or above, running on an IBM PC or compatible computer.

See Section 7.2, “Rehosting” [52] for information about additional host computers.

---

## *2.4. Target Configurations*

For M68K Ada Version 1.7, the standard target is the XGC simulator. The generated program may be located in a Boot PROM,

---

and copied to RAM for execution, or may be executed directly from PROM.

The run-time system is easily customized for hardware targets, including any of the Motorola MVME series single board computers that have a M68000 family CPU.

---

## *2.5. Target Operating System*

No target operating system is required since M68K Ada includes all the necessary run-time system functions to support application programs running on a bare target board.

---

## *2.6. Programming Support Environment*

M68K Ada includes a tool to automatically compile and recompile a given Ada program. It runs the Ada compiler as necessary to build a given program, while enforcing the Ada rules about dependencies among compilation units.

In addition, the programming support environment consists of the standard GNU/UNIX software development tools, which provide configuration management, automated program configuration and construction, automated regression testing, and much more.

We recommend the Bash shell since it offers conformance to the POSIX standard, and supports command line working. Bash is not included with M68K Ada, but is available from any GNU site.

---

## *2.7. Host-Target Communication*

Two methods are available for transferring data from the host to the target. At the host the following facilities are provided:

- A standard RS-232-C port connected to UNIX terminal interface
  - A TCP/IP network connection
-

Either of these communication standards can be used provided that a compatible capability is available on the target.



---

The *Ada 95 Reference Manual*, ANSI/ISO/IEC-8652:1995, explicitly allows variations between Ada processors in a number of aspects. This chapter describes the language supported by M68K Ada and is organized according to the appropriate chapters and annexes of the Ada Manual.

---

### *3.1. Overview*

M68K Ada supports several restricted Ada 95 profiles that prohibit the use of unsafe language features, and which are compatible with the requirements for high-integrity software applications.

- The XGC profile (the largest profile and the default)
- Ravenscar (which includes a limited form of tasking)
- Restricted run-time system (for ARINC 653 applications)
- No run-time system (for safety-critical applications)

Language features that are always restricted are not supported at all. This means that the compiler and run-time system can be optimized for the safe subsets and unlike unrestricted compilers, need not be hindered by the need to support complex and inefficient features that are never used.

The gain in efficiency is evident in the performance figures, which are an order of magnitude smaller and faster than competing compilation systems that support the full language.

The following list gives language features that are prohibited. The references to sections in this list apply to the *Ada 95 Reference Manual*. Further details appear in the respective section below:

<b>Feature</b>	<b>Ada RM Section</b>
Partitions of Distributed Systems	Annex E
Exception propagation	Sections 3.1 and 11
Finalization in packages	Section 7.6
Some predefined packages	Annex A
Streams	Section 13
Class-wide operations with tagged types	Section 3.9

---

### *3.2. Section 2: Lexical Elements*

Ada source programs are represented in standard text files, using Latin-1 coding. Latin-1 is ASCII with additional characters used for representing foreign languages. The lower half (character codes 16#00# ... 16#7F#) is identical to standard ASCII coding, but the upper half is used to represent the additional characters. Any of these extended characters is allowed in character or string literals. Moreover, extended characters that represent letters can be used in identifiers.

On the target M68K Ada supports the character sets defined by the *Ada 95 Reference Manual*. These are the predefined types `Character` and `Wide_Character`.

The predefined type `Character` is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO 10646 Basic Multi-lingual Plane (BMP).

The predefined type `Wide_Character` is a character type whose values correspond to the 65536 code positions of the ISO 10646 Basic Multi-lingual Plane (BMP).

The maximum number of characters in a source line is 255.

The maximum length of a lexical element is 255 characters.

---

### *3.3. Section 3: Declarations and Types*

Declarations and types are supported as specified in the Ada standard except for tagged types (See RM Section 3.9).

---

#### **3.3.1. Uninitialized Variables**

When the compile time options `-Wuninitialized -0` are used, the compiler will flag variables that may be uninitialized.

---

#### **3.3.2. Enumeration Types**

Enumeration types are supported as defined in the *Ada 95 Reference Manual*. Additional code and read-only data are generated to support the attributes `'Image`, `'Pos` and `'Val`.

The size of enumeration objects is the minimum required to accommodate all the values, and including any representations given in a representation clause. The compiler will select a size of 8, 16 or 32 bits as appropriate.

Enumeration types may be packed to reduce wasted space in arrays of enumeration objects.

### 3.3.3. Integer Types

M68K Ada provides five predefined Integer types:

- the type `Short_Short_Integer`
- the type `Short_Integer`
- the type `Integer`
- the type `Long_Integer`
- the type `Long_Long_Integer`

Table 3.1, “Attributes of the Predefined Integer Types” [12] gives the values of the attributes `Size`, `First` and `Last` for these types.

**Table 3.1. Attributes of the Predefined Integer Types**

Type name	Size	First	Last
<code>Short_Short_Integer</code>	8	$-2^7$	$2^7-1$
<code>Short_Integer</code>	16	$-2^{15}$	$2^{15}-1$
<code>Integer</code>	32	$-2^{31}$	$2^{31}-1$
<code>Long_Integer</code>	32	$-2^{31}$	$2^{31}-1$
<code>Long_Long_Integer</code>	64	$-2^{63}$	$2^{63}-1$

**User-Defined Types.** For a user-defined integer type, the compiler automatically selects the smallest compatible predefined integer type as the base type. For example, given the following type definition:

```
type My_Integer is
range -10 .. +10;
```

the compiler will use `Short_Short_Integer` as the base type, and `My_Integer'Size` will be 8 bits.

**Modular Types.** M68K Ada supports modular types up to 64 bits in size. Like the integer types, these are represented in 8, 16, 32 or 64 bits as appropriate. The following declarations are legal:

```
        type word_8 is
          mod 256;
type word_16 is
          mod 65536;
type word_32 is
          mod 2**32;
type word_64 is
          mod 2**64;
```

The standard Ada 95 operators for modular types are supported.

---

### 3.3.4. Floating Point Types

M68K Ada provides four predefined floating-point types:

- the type `Short_Float`
- the type `Float`
- the type `Long_Float`
- the type `Long_Long_Float`

The types `Short_Float` and `Float` are represented by the 32-bit single precision IEEE format; the types `Long_Float` and `Long_Long_Float` are represented by the 64-bit IEEE format. Note that the IEEE 80-bit extended precision format is not supported by M68K Ada.

Table 3.2, “Basic Attributes of Floating Point Types” [14] gives the values of the attributes for the predefined floating-point types.

**Table 3.2. Basic Attributes of Floating Point Types**

Attribute	Float	Long_Float
Size	32	64
Digits	6	15
Machine_Radix	2	2
Machine_Mantissa	23	52
Machine_Emax	128	1024
Machine_Emin	-125	-1021
Machine_Rounds	False	False
Machine_Overflows	False	False

### 3.3.5. Fixed Point Types

M68K Ada supports fixed-point types up to 64 bits in size using 8, 16, 32 or 64 bits as appropriate. The value of 'Small may be either a power of two (chosen by the compiler), or an arbitrary value given in a representation clause.

## 3.4. Section 4: Names and Expressions

Names and expressions are fully supported in the default profile.

Static expressions of the type `universal_integer` or `universal_real` have no limit on the implemented range or precision. Evaluation of such expressions is carried out by a general universal arithmetic package.

Non-static expressions of type `universal_integer` are evaluated at run time using the smallest predefined integer type with sufficient range.

If run-time floating point support is available, non-static expressions of type `universal_real` are evaluated at run time using 64-bit double-precision floating point.

### 3.5. Section 5: Statements

Some task-related statements are prohibited. All other statements are supported as described in the *Ada 95 Reference Manual*.

The prohibited statements are:

- terminate alternative for selective wait
- abort
- requeue

---

### 3.6. Section 6: Subprograms

Subprograms are fully supported.

---

### 3.7. Section 7: Packages

Except for finalization, which depends on tagged types, packages are fully supported.

---

### 3.8. Section 8: Visibility Rules

Visibility rules are fully supported.

---

### 3.9. Section 9: Tasks and Synchronization

Tasks, protected types and task-related statements are permitted subject to any explicit restrictions.

- Task declarations are only permitted at the library level. Tasks may not be dynamically allocated. Tasks may not terminate.
- Protected objects are only permitted at the library level. Protected objects may not be dynamically allocated. The maximum number of entries for a protected object is one. The entry barrier must

be a simple Boolean variable, and a maximum of one task may wait on the entry.

- The package `Ada.Real_Time` is provided, and the type `Ada.Real_Time.Time` may be used in a delay until statement.
- The package `Ada.Synchronous_Task_Control` is provided and offers an alternative and possibly more efficient way for tasks to communicate.
- The Ada 83 rendezvous is supported except for the terminate alternative.

Except for the restrictions on the number of tasks in an entry queue and the nested rendezvous (which are checked at run time), the compiler will reject any program that does not conform to the default or given Profile.

---

### 3.9.1. type Duration

The predefined type `Duration` is 32-bit fixed-point type. The value of the least significant bit is one microsecond. Table 3.3, “Attributes of the Predefined Type Duration” [16] gives the attributes of this type.

**Table 3.3. Attributes of the Predefined Type Duration**

Attribute	Value	Comment
Delta	1.0E-6	One microsecond
Small	1.0E-6	One microsecond
First	-2147.483648	Approx. -35 minutes
Last	2147.483647	Approx. 35 minutes

The types `Time` and `Time_Span` from predefined package `Ada.Real_Time` have the same representation as type `Duration`. However the type `Time` is declared as a modular type and comparisons of two times correctly account for the 71-minute cycle.



### 3.9.2. Shared Variables

M68K Ada supports the pragma `Volatile`, which guarantees that a variable will be fetched from memory each time it is referenced, and will be stored in memory on each assignment.

M68K Ada also supports the pragmas `Atomic`, `Atomic_Components`, and `Volatile_Components`, as specified in Section C.6 of the *Ada 95 Reference Manual*.

---

## 3.10. Section 10: Program Structure and Compilation Issues

An M68K Ada program may use any mixture of programming languages supported by the compiler, assembler or the linker. One procedure must become the main program, but this need not be written in Ada 95.

If the main program is written in Ada then it must be a parameter-less library procedure. If the main program is written in C then the standard arguments of function `main` will be null.

The main program is called by a run-time system module (`art0.S`) that initializes the stack and variable data area, and which contains code to handle traps and interrupts. Code in `art0.S` can also copy program sections from the boot PROM into RAM.

For the M68K Microprocessor, the entire program will consist of four items:

- The startup module, `art0.S`, which contains the entry point
- The function `main`, which calls any Ada elaboration routines then calls the Ada main procedure
- The Ada program comprising the Ada main procedure and any library packages in the link closure of the main program
- Library routines as required to support the generated code (multiply and divide for example)

The ANSI C libraries `libc` and `libm` may also be used via appropriate import pragmas.

---

### 3.11. Section 11: Exceptions

Exceptions may be declared and raised as described in the Ada 95 standard. However exception handlers can only handle exceptions raised locally. The propagation of exceptions is not supported.

The predefined exceptions `Program_Error`, `Numeric_Error` and `Constraint_Error` are raised under the conditions given in the Ada 95 Standard.

The predefined exception `Storage_Error` is raised by an explicit raise statement, or when entering a subprogram, or when allocating the stack space for a data object or task declaration. The additional code for these checks is generated by default.

---

### 3.12. Section 12: Generic Units

Generic Units are supported as defined in the *Ada 95 Reference Manual*.

---

### 3.13. Section 13: Representation Issues

M68K Ada supports all of the implementation-dependent features of *Ada 95 Reference Manual* Section 13 that have a useful meaning in an embedded system.

In particular:

- The pragma `Pack` is supported.
  - Length clauses are supported, including the following:
    - Size specification for types
    - Small specification for fixed point types, using arbitrary values
-

- Storage\_Size specification for tasks
- Enumeration representation clauses are supported.
- Record representation clauses are supported.
- Alignment clauses are supported (up to the maximum data object size).
- Address clauses are supported for constants and variables.
- The pragma Interface is supported.
- Unchecked programming is supported.
- The predefined package Machine\_Code is supported.

The following are *not* supported:

- interrupt entries for tasks
- address clauses for subprograms, packages or tasks
- the predefined packages Ada.Unchecked\_Deallocation and Unchecked\_Deallocation
- the predefined package System.Storage\_Pools
- the predefined package Ada.Streams

---

### **3.13.1. Definitions from the predefined package System**

Table 3.4, “Named Numbers from package System” [19] specifies values from the predefined package System.

**Table 3.4. Named Numbers from package System**

<b>Named Number</b>	<b>Value</b>
Min_Int	$-2^{63}$
Max_Int	$2^{63} - 1$
Max_Binary_Modulus	$2^{64}$

<b>Named Number</b>	<b>Value</b>
Max_Nonbinary_Modulus	32767
Max_Base_Digits	15
Max_Digits	15
Max_Mantissa	63
Fine_Delta	$2.0^{-63}$
Tick	1.0 Microseconds

---

### **3.13.2. The type Address**

The predefined type Address is 32 bits in size, and the unit of storage addressed is an 8-bit byte. The value of the null address is zero. Address is declared in the visible part of package System, so that address expressions may contain numeric literals.

---

## *3.14. Input-Output*

The packages `Ada.Text_IO`, `Ada.Sequential_IO` and `Ada.Direct_IO` require support from the system call interface. When running on the target simulator, the system call interface is supported using the host operating system, and, for example, a call to open a file will open a host file. When the application is running on the target computer, a system call handler may be supplied that supports the calls with an IO system. An example of such a handler is included in the run-time system.

The package `Ada.Storage_IO` is supported as described in the Ada 95 Reference Manual.

---

## *3.15. Annex A: Predefined Language Environment*

The following predefined library units are provided.

- package Ada
- package Ada.Asynchronous\_Task\_Control

- package Ada.Characters
- package Ada.Characters.Handling
- package Ada.Characters.Latin\_1
- package Ada.Characters.Wide\_Latin\_1
- package Ada.Decimal
- package Ada.Dynamic\_Priorities
- package Ada.Interfaces
- package Ada.Interrupts
- package Ada.Interrupts.Names
- package Ada.IO\_Exceptions
- package Ada.Numerics (not all child packages are supported)
- package Ada.Real\_Time
- package Ada.Strings (not all child packages are supported)
- package Ada.Synchronous\_Task\_Control
- package Ada.Task\_Identification
- package Ada.Text\_IO
- package Ada.Unchecked\_Conversion
- package IO\_Exceptions
- package Interfaces (not all child packages are supported)
- package Machine\_Code
- package System
- package System.Address\_to\_Access\_Conversions

- package System.Machine\_Code
  - package System.Storage\_Elements
  - function Unchecked\_Conversion
- 

### *3.16. Annex B: Interface to Other Languages*

Annex B is partially supported. In particular, the predefined package Interfaces is supported.

The following list gives language features that are *not* available:

<b>Feature</b>	<b>Reason for restriction</b>
Interfaces.COBOL	Not Applicable
Interfaces.FORTRAN	Not Applicable

---

### *3.17. Annex C: Systems Programming*

Annex C is supported as follows.

C1. Access to Machine Operations  
Section C1 is fully supported.

C2. Required Representation Support  
Section C2 is fully supported.

C3. Interrupt Support  
Interrupts are fully supported. In particular, the package Ada.Interrupts.Names is customized for the target computer.

C4. Prelaboration Requirements  
Section C4 is fully supported.

C5. Pragma Discard\_Names  
Section C5 is fully supported.

C6. Shared Variable Control  
Section C6 is fully supported.

C7. Task Identification and Attributes

Section C7 is not fully supported because of the restrictions on tasking.

---

### 3.18. Annex D: Real-Time Systems

Annex D is mostly supported and meets the requirements of the several profiles.

However, the restrictions defined here and in Annex H are supported, and used as defaults, as described in Appendix B, *Restrictions and Profiles* [65].

D1. Task Priorities

Section D1 is fully supported with subtype Priority having a range from 0 .. 127, and subtype Interrupt\_Priority having a range from 128 .. 255.

D2. Priority Scheduling

Section D2 is fully supported with the task dispatching policy FIFO\_Within\_Priorities.

D3. Priority Ceiling Locking

Section D3 is fully supported with Ceiling\_Locking.

D4. Entry Queuing Policies

For protected types, section D4 is restricted so that the maximum queue length is one.

D5. Dynamic Priorities

The features of Section D5 are supported by default and may be prohibited by the use of appropriate restrictions or profiles.

D6. Preemptive Abort

The features of Section D6 are prohibited.

D7. Tasking Restrictions

Section D7 is fully supported, and includes new restrictions.

D8. Monotonic Time

Section D8 is fully supported.

D9. Delay Accuracy

Section D9 is fully supported.

D10. Synchronous Task Control

Section D10 is fully supported.

D11. Asynchronous Task Control

The features of Section D11 are supported by default and may be prohibited by the use of appropriate restrictions or profiles.

D12. Other Optimizations

The requirements of Section 12 are met.

---

### *3.19. Annex E: Distributed Systems*

Only a single partition is available.

---

### *3.20. Annex F: Information Systems*

Annex F is not supported.

---

### *3.21. Annex G: Numerics*

Annex G (complex numeric types) is not supported.

---

### *3.22. Annex H: Safety and Security*

The restrictions defined here and in Annex D are supported, and used as defaults, as described in Appendix B, *Restrictions and Profiles* [65].

The Ravenscar profile requires several new restrictions, which are also supported.



### 3.23. Annex J: Obsolescent Features

This Annex is almost completely supported. The only missing language feature is the predefined package `Unchecked_Deallocation`, which cannot be supported because the run-time system has no means of freeing allocated memory.

---

### 3.24. Annex K: Language-Defined Attributes

Annex K is partially supported. See discussion in other sections.

---

### 3.25. Annex L: Language-Defined Pragmas

The language-defined pragmas in Annex L are fully supported.



# *User Interface and Debugging Facilities*

---

This chapter summarizes how to use the cross compiler. The management of compilation, cross-development and debugging are described briefly.

---

## *4.1. Compiler Invocation*

M68K Ada uses the UNIX shell command line interface. The compiler, the tools and the libraries are systematically named, and installed in a formal directory structure. This permits different versions of M68K Ada to be installed and used at the same time, without confusion over which files belong to which version. For the compiler, the convention is this. The native compiler is called `gcc`, and is located in the directory `/usr/bin`, or in `/usr/local/bin`. Cross compilers have a different name, and are installed under `/opt/m68k-ada-1.7/`, for example, with the executable images in the directory `/opt/m68k-ada-1.7/bin/`.

The main program of the M68K Ada compiler is called `m68k-coff-gcc` and is located in the directory

`/opt/m68k-ada-1.7/bin/`. Depending on which source files and command line options are given, the main program will call the Ada compiler, C compiler, assembler, or linker.

The other executable images are named in the same way. For example, the native assembler is called `as`; the cross assembler is called `m68k-coff-as`.

---

## 4.2. *Compilation*

For Ada 83 and Ada 95 the predefined program library is located in a standard place, which is target dependent. The location is `/opt/m68k-ada-1.7/lib/gcc-lib/m68k-coff/2.8.1/adalib/`.

Ada source files are always compiled in the context of a program library. While M68K Ada does not have a closed library as some other Ada compilers do, it does generate and use library file information. By default this goes in the current directory. Library files use the `.ali` (Ada library) suffix. Also M68K Ada requires each Ada compilation unit to be in a separate file with the file name the same as the unit name. There must be a file extension which is `.ads` for a package or subprogram specification and `.adb` for a body.

Where a source file contains more than one compilation unit, then the program `m68k-coff-gnatchop` may be used to divide the file. This program will write the output files in the current directory, or (more usefully) into a named directory.

For example:

```
bash$  
m68k-coff-gnatchop big-file.ada src
```

---

### 4.2.1. **Format and Content of User Listings**

The compiler accepts several command line options to control the format of listings. By default, no listing is generated at all.

-gnatl

Output full source listing with embedded error messages.

-gnatv

Verbose mode. Full error output with source lines to stdout.

-gnatk

Keep going despite syntax errors.

```
bash$
m68k-coff-gcc -gnatlqv ackermann.adb
XGC Ada m68k-ada/-1.7/
Copyright 1992-2002 Free Software Foundation, Inc.

Compiling: ackermann.adb (source file time stamp: 2001-04-25 16:57:54)

1. function Ackermann (m, n : Integer) return Integer is
2. begin
3.   if m = 0 then
4.     return n + 1;
5.   elsif n = 0 then
6.     return Ackermann (m - 1, 1);
7.   else
8.     return Ackermann (m - 1, Ackermann (m, n - 1));
9.   end if;
10. end Ackermann;
11.

11 lines: No errors
```

The assembler can also generate a listing, as follows:

```
bash$
m68k-coff-gcc -Wa,-a -O -c ackermann.adb
1          .file  "ackermann.adb"
2          gcc2_compiled.:
3          __gnu_compiled_ada:
4          .text
```

```
5                                     .even
6                                     .globl _ada_ackermann
7                                     _ada_ackermann:
8 0000 4E56 0000                       link.w  %a6,#0
9 0004 2F0A                             move.l  %a2,-(%sp)
10 0006 226E 0008                       move.l  8(%a6),%a1
11 000a 206E 000C                       move.l  12(%a6),%a0
lots of output...
```

The `objdump` program can also generate a listing by disassembling the object code.

```
bash$
      m68k-coff-objdump -d ackermann.o
ackermann.o:      file format coff-m68k

Disassembly of section .text:

00000000 <_ada_ackermann>
0:  4e56 0000      linkw  %fp,#0
4:  2f0a         movel  %a2,%sp@-
6:  226e 0008     moveal %fp@(8),%a1
a:  206e 000c     moveal %fp@(12),%a0
...
```

---

### 4.3. Errors and Warnings

No object code is generated for units that contain errors.

There are three levels of messages:

- Fatal errors, where the compiler is unable to continue
- Errors, which explain the nature of the error

- Warnings, which are less severe than errors, and which do not prevent code generation

If the listing option is set, then the messages will be correctly placed in the listing, with a pointer to the lexical token relating to the message.

---

#### *4.4. Other Software Supplied*

M68K Ada includes a number of other tools to support software development, as follows:

`m68k-coff-addr2line`

which converts given target addresses to source file line numbers

`m68k-coff-ar`

which is used to build object code libraries

`m68k-coff-gdb`

which is the symbolic debugger

`m68k-coff-gnat chop`

which may be used to divide a file that contains more than one compilation unit into one file for each unit

`m68k-coff-gnat find`

which is used to find Ada symbols in source files

`m68k-coff-gnat ls`

which is used to list Ada units

`m68k-coff-gnat make`

which uses the Ada rules to automatically compile, recompile and build an Ada program

`m68k-coff-gnat prep`

which is an Ada pre-processor

`m68k-coff-gnat psta`

which prints the target package Standard

m68k-coff-gnatpsys

which prints the target package System

m68k-coff-gnatxref

which is the Ada cross reference tool

m68k-coff-nm

which lists the symbols from object files

m68k-coff-objcopy

which is used to copy and reformat object code files

m68k-coff-objdump

which is used to dump information from object code files and includes an option to disassemble

m68k-coff-ranlib

which generates an index to the contents of an archive and stores it in the archive

m68k-coff-run

which is the simulator

m68k-coff-sim

which is an interactive simulator

m68k-coff-size

which prints the size of an object code or executable file

m68k-coff-strings

which lists debug symbols and other strings in an object code file

m68k-coff-strip

which removes debug symbol table information from object code files



## 4.5. Debugging Facilities

The GNU debugger, **gdb**, as customized for M68K Ada, offers many features for debugging both at the high-level language level and with machine code.

Using the host-target link and the XGC monitor, the debugger can debug programs running on a remote target.

M68K Ada includes a target simulator that accurately simulates the Motorola M68000 family instruction set. The simulator includes the following features:

- Simulates the entire M68000 instruction set, with an option to specify the CPU type (MC68000, MC68010...)
- Simulates approximate instruction timing, including wait states
- Prints statistics giving execution time, number of instructions in each class, number of nullified instructions
- Prints current task state information. Traces task and interrupt states for a time range with several sorts of trigger.
- Prints test coverage information, either for the whole program or for a given source file



# *Performance and Capacity*

---

This chapter describes host performance and capacity, and target code performance of M68K Ada.

---

## *5.1. Host Performance and Capacity*

The compile time performance of M68K Ada is generally very good. For example, one application, which consists of 20,000 lines of Ada 95, compiles in 16 seconds of CPU time, on a 133MHz Pentium UNIX system. That is a rate of 60,000 lines per minute. These times are for compilations using the highest optimization level.

Ada package specifications, which typically involve very little generated code, compile very quickly. On the other hand, extensive use of generic instantiations or in-line expansion, which can result in large amounts of generated code, can greatly reduce the line-per-minute rate.

Because of the overhead of loading the compiler, the line-per-minute rate will be bigger for a large compilation unit than for a small one. Also once the compiler is loaded, further compilations will proceed a lot faster.

The Ada compiler builds a compact tree structure in memory for each compilation unit. Clearly the size of the tree will depend on the size of the unit, but experience suggests that a UNIX system with 16M bytes of real memory is more than adequate for typical program development, even where X-Windows and Motif are used. However where very large Ada units are to be compiled, 24M bytes of memory will be a better size, and 32M bytes should be sufficient for even the largest compilations. No use is made of any previous compilation of the same unit to increase compilation speed.

---

## 5.2. Target Code Performance

The target code performance of M68K Ada is generally very good. The compiler generates code that compares well with other compilers, and which the assembly language programmer would find difficult to beat. See the examples of generated code in Appendix A, *Examples of Generated Code* [59].

The results of running the three benchmark programs *Sieve*, *Ackermann* and *Whetstone* are given in Table 5.1, “Benchmark Results” [36]. These programs were run on a simulated MC68040, with a 25 MHz clock.

**Table 5.1. Benchmark Results**

<b>Benchmark</b>	<b>Result at 25 MHz</b>
Ackermann	503 mSec
Sieve	232 mSec
Whetstone	3450 KWIPS

Table 5.2, “Task-Related Metrics” [37] gives timings for several task-related features. The clock frequency is 25 MHz.

**Table 5.2. Task-Related Metrics**

Metric	Time in Microseconds
Interrupt latency (C.3.1 (15))	tbd
From call of trivial protected procedure to return from entry	tbd
Call of Clock (D.8 (44))	3
Lateness of a delay (D.9 (13))	50
Suspend_Until_True, where state is already True	8
Set_True to return from Suspend_Until_True	55
Trivial protected procedure call (D.12 (6))	6

---

### 5.2.1. Optimization and Code Quality

M68K Ada uses many traditional optimizations to improve the size and execution speed of the generated code. The following list includes some of the optimizations.

- Sub-expression commoning
- Loop unrolling
- Loop variable induction
- Strength reduction
- Constraint check elimination
- Loop invariant hoisting
- Load and store elimination
- Register allocation
- Unreachable code elimination

- Tail recursion optimization

The overall level of optimization is controlled by the `-O` option. The default is optimization level 2. Also many of the optimizations are tied to a further compile-time option and can be enabled or disabled as necessary.

---

### 5.2.2. Constraint Checks

In general, constraint checks are eliminated wherever possible, and constraint check expressions are subject to all the usual optimizations.

Most redundant checks are eliminated. In the example that follows, constraint checks such as those at (1), (2) and (3) are generally eliminated.

```
I : Integer range -2 .. 2;
J : Integer range 0 .. 10;

type BT is
    access T;
V : BT;

I := 22 mod 3;    -- (1) no checks needed at run time
I := J;           -- (2) check on top limit only
V := new T (...);
if V.L = ... then -- (3) no null access check
    -- (4) current variant is correct
```

In the example shown, the run-time checks performed are as follows:

- A check on the top limit only is performed for (2).
- A discriminant check is performed for (4).

### 5.2.3. Space for Unused Variables

No space is allocated for scalar variables that are unused. Space for arrays and records is always allocated.

---

### 5.2.4. Space for Unused Subprograms

Subprograms that are declared in a package but unused in a program are always loaded if the package is loaded.

---

### 5.2.5. Evaluation of Static Expressions

Static expressions are always evaluated according to the rules of the *Ada 95 Reference Manual* Section 4.8. Other compile-time-constant expressions may be evaluated at compile time too.

---

### 5.2.6. Elimination of Unreachable Code

In most cases code that is unreachable is eliminated.

---

### 5.2.7. Common Sub-expressions

In the following code example, the address of the element of the array is computed once.

```
A(I) := A(I) + 1;
```

---

### 5.2.8. Loop Invariants

In the following matrix code, the address of the element A(I, J) is computed for the first iteration, then for subsequent iterations the address is incremented by the size of the element.

```
                for I in 1 .. N loop
                for J in 1 .. M loop
    A (I, J) ...
end
                loop;
end
                loop;
```

---

### **5.2.9. Bound Checks**

In general, redundant array bounds checks are eliminated.

---

### **5.2.10. The pragma Inline**

The pragma Inline is supported, except where the subroutine mentioned in the pragma is ineligible. Inlining across compilation units may be disabled using a compile-time option.

---

### **5.2.11. Procedure Calling Overhead**

As an example of the subprogram calling overhead, the code sizes for Ackermann's function are as follows:

- Total code size for Ackermann's function = 66 bytes, or 72 bytes with stack overflow check.
- Instructions executed per call = 20, or 22 with stack overflow check.

---

### **5.2.12. The Rendezvous**

In a rendezvous, the accept statement body is executed by the owning task, never by the calling task. No tasking optimizations are performed but the special case of a null accept statement is handled separately.



### 5.2.13. Space Requirements

For a task 114 bytes are allocated for the task control block. In addition, there are 14 bytes for each task entry. The stack size is either the default size of 4096 bytes, or the value given in the task type's length clause.

The space overhead for a protected object is 25 bytes.

The size of a null program is approximately 2.5K bytes. The size of a minimal program that uses tasking (tasks, protected objects and delay statements) is approximately 7K bytes. These sizes include code, read-only data and variables, but exclude stack space.



# *Cross-Compiler and Run-Time Interfacing*

---

The internal structure of the M68K Ada cross-compiler and run-time system are described in this chapter.

---

## *6.1. Cross-Compiler Issues*

The following sections describe the design of the native and cross compilers in general, and provide a more detailed description of the M68K Ada compiler.

---

### **6.1.1. Background**

The M68K Ada compiler uses the front end of the GNAT compiler from New York University. This compiler was developed with funding from the United States Department of Defense to be the compiler promised in the Ada requirements document known as *Steelman*.

GNAT consists of an Ada 95 front end, a code generator, and a middle phase that translates the Ada program into the intermediate language used by the code generator. The code generator is taken from GCC, the GNU C Compiler, as are the other tools required to complete the compilation system.

The Free Software Foundation designed GCC to be the compiler of the GNU UNIX-like operating system, and was required to support the ANSI C programming language and work with other UNIX tools. It was also required to generate high-quality code for any computer that could be expected to run UNIX.

These requirements led to the implementation of a compiler that became an obvious base for other programming languages, and today GCC supports C++, Objective C, Pascal, Modula-3, FORTRAN, and Ada.

GCC has also been developed to meet the needs of embedded system programmers, and can be configured as a cross compiler using a minimal run-time system. The GNAT Ada front end is the most complete implementation of the Ada 95 language available. Most of the optional features are supported, including the distributed systems Annex and the safety-critical Annex.

---

## *6.2. Compiler Phase and Pass Structure*

The compiler, the assembler and the linker are three separate programs, but are normally run under the control of a small driver program, `gcc`. Given compile-time options, and a source file, `gcc` uses a target-dependent specification file to determine which passes are required. These are then run using UNIX pipes or temporary files to pass data between the separate programs. Many of the defaults can be overridden with compile-time options.

The default for the `gcc` command is to use the latest version of the native compiler. For the M68K Ada compiler a further driver program is supplied. This is called `m68k-coff-gcc` and runs the compiler, assembler and linker targeted to the M68K Microprocessor rather than the native ones. Either driver can run an earlier version of the compiler, if installed.

The compiler has a language-dependent front end, which builds internal representation of the program being compiled, then calls the target-dependent code generator to generate assembly language. The M68K Ada compiler includes front end for ANSI C as well as Ada 95.

The Ada front-end comprises four phases, which communicate by means of a compact *Abstract Syntax Tree* (AST). The implementation details of the AST are hidden by several procedural interfaces that provide access to syntactic and semantic attributes. The layering of the system, and the various levels of abstraction, are the obvious benefits of writing in Ada, in what one might call “proper” Ada style.

The back end generates code for the M68K Microprocessor and includes phases to handle optimizations, register allocation and code generation. The code generator uses a pattern matching technique to ensure good use of the target computer's instruction set.

---

## 6.3. *Compiler Module Structure*

---

### 6.3.1. **Intermediate Program Representations**

The compiler generates assembly language, which is automatically passed to the assembler. The assembler generates object code, and several different object code formats are supported. The utility program objcopy may be used to change the format among any of those supported.

---

### 6.3.2. **Final Program Representation**

The final program representation is one of a number of industry-standard formats, including but not limited to the following:

- COFF (default)
- Motorola S-Records

- Intel Hex
- Tek Hex

The default format is COFF, which can include symbolic information to help with debugging. When COFF files are converted into the other formats, some or all of the debugging information is lost.

---

### **6.3.3. Compiler Interfaces to Other Tools**

M68K Ada provides information for other tools— notably the GNU debugger GDB and the GNU profiler GPROF. GPROF is not included with M68K Ada, but may be used with a native Ada compiler to provide a useful analysis of software that is intended to be run on the target microprocessor. M68K Ada can also provide information for future program analysis tools. This is done by an implementation-defined pragma that allows the programmer to annotate the Ada source with arbitrary comments that are preserved in the internal data structures.

---

## *6.4. Compiler Construction Tools*

Technically, the crucial asset of the GCC is its mostly language-independent, target-independent code generator. It produces code of excellent quality both for CISC machines such as the Intel and Motorola families, as well as RISC machines such as the SPARC. The machine dependencies of the code generator represent less than 10 per cent of the total code.

To add a new target to GCC, an algebraic description of each machine instruction must be given using a register-transfer language. Most of the code generation and optimization then uses the RTL, which GCC maps when needed into the target machine language. Furthermore, GCC produces high-quality code, comparable to that of the best compilers.

## 6.5. *Installation*

M68K Ada is shipped on CD-ROM. As with most UNIX software, installation is simple. For Solaris, M68K Ada is shipped as a Solaris package that is installed using the Solaris `pkgadd` command. For Linux M68K Ada is supplied as one or more compressed tar format files. To install, enter the appropriate tar command then follow the enclosed installation instructions. Note that installation requires access to directories that may be under the control of the system administrator.

---

## 6.6. *Run-Time System Issues*

M68K Ada includes a run-time system that supports C and Ada. This includes the basic functions that are common to both languages, such as program startup and interrupt handling. In addition, each language is supported by a number of standard libraries, as required by the language definition.

---

### 6.6.1. **The Stack**

The Ada main program is given a stack, where the location and size are determined by the linker script file. The stack is used to support subprogram calls, and typically contains a linked sequence of stack frames that contain saved registers and subprogram data.

Each task has a stack that is allocated at elaboration time from the free memory declared in the linker script file. If insufficient free memory is available, then the predefined exception `Storage_Error` is raised.

Interrupt handlers use a separate stack also declared in the linker script file.

---

### 6.6.2. **Subprogram Call and Parameter Handling**

The subprogram calling convention follows the Motorola standard with parameters passed on the stack.

**Register saving.** M68K Ada uses the callee-save convention for saving registers across subprogram calls. On entry to a subprogram, the registers of the calling subprogram that are modified locally are saved in the current stack frame. On return from a subprogram, the registers are restored.

**Parameter passing.** Parameters are passed on the stack.

**The call instruction.** M68K Ada uses the JSR instruction to call a subprogram.

**Subprogram entry.** For subprogram entry, the compiler generates code to establish a new stack frame. This may include code to check for stack overflow. The compiler is able to recognize several special cases where the worst-case code can be improved. In particular, for leaf<sup>1</sup> subprograms that have no need for a stack frame data, the stack frame is completely eliminated and the code to set up the frame, and remove it on exit, is not generated.

**Subprogram exit.** For subprogram exit, the compiler generates code to remove the current stack frame, and return to the calling subprogram.

**The return value.** Function values are returned in a register if possible. If not then the calling subprogram allocates space in its stack frame then passes the address of the space to the called subprogram, which copies the function value to that address.

---

### 6.6.3. Data Representation

The following table shows the number of bits in the data representation for the M68K Microprocessor.

---

Type	M68K Microprocessor
Integer	8, 16, 32 and 64 signed
Modular	8, 16, 32 and 64 unsigned
Fixed	8, 16, 32 and 64 signed

---

---

<sup>1</sup>A leaf subprogram is one that makes no subprogram calls.



<b>Type</b>	<b>M68K Microprocessor</b>
Floating Point	32 and 64
Enumeration	8, 16, 32 and 64

---

Storage allocation for array types is simply the number of components multiplied by the allocation for each component. Components can be packed and bit aligned in some cases. Unconstrained arrays have a descriptor with lower and upper bounds for each index. Note that dynamically unconstrained arrays are prohibited.

Storage allocation for record types is the sum of the individual component allocations, which are byte aligned by default. Components can be packed and bit aligned in some cases.

The pragma Pack causes pack-able array and record components to be allocated in adjacent bits without regard to byte boundaries.

---

#### **6.6.4. Implementation of Ada Tasking**

M68K Ada supports a limited form of Ada tasking that permits static tasks, protected types and a limited form of rendezvous. The features supported may be further restricted by use of individual restrictions, or by the pragma Profile.

The general strategy is for the compiler to translate Ada tasking operations into run-time system calls, using data types from the predefined package `XGC.Tasking`.

Some language features (delays for example) are supported by child subprograms.

In addition the package `XGC.Preemption_Control` is required to give the run-time system exclusive access to the tasking data structures.

The above packages are only included in an application program if the corresponding language features are used. A null program is linked with only the minimal run-time system module `art0.S`.

## 6.7. Exception Handling System

M68K Ada supports exception declarations, the raise statement, and exception handlers. It does not support exception propagation. We expect M68K Ada application programs to regard an exception as a fatal error, and to log the context of the failure (in non-volatile RAM for example), then to restart the program.

There is no overhead associated with calling or entering a subprogram in which an exception is declared, other than the space required to hold the exception descriptor. This a small record that contains the name of the exception (as a string), and several other items required to satisfy the needs of the predefined package `Ada.Exceptions`.

An exception may also be raised by a call of `Ada.Exceptions.Raise_Exception`. The advantage of making the call rather than using the raise statement is that the call may attach a message to the exception.

Unhandled exceptions, hardware faults and deadline errors are reported within the run-time system, and can be handled as interrupts. The default action is to log the fault (via application-dependent code), then do a warm restart.

---

## 6.8. I/O Interfaces

The predefined library packages `Text_IO` and `Ada.Text_IO` are partially supported so that test programs can write their results to an output stream.

For application program input and output, it is necessary to use low-level features such as representation clauses and package `Machine_Code`.

---

## 6.9. Documentation

M68K Ada includes comprehensive electronic documentation for the compiler, the tools, and the Ada programming language.

# *Re-targeting and Re-hosting*

---

M68K Ada is shipped in binary format and source format. The binary version is created for a specific host computer (for example a Sun SPARC running Solaris 2.6) and for a specific target computer (the M68K Microprocessor) and will only run on that host for that target.

The source version consists of the standard GCC distribution, with the new code generator, assembler, disassembler etc., the relevant GNAT baseline, and run-time software written for M68K Ada.

---

## *7.1. Retargeting*

M68K Ada is a customization of the GCC compiler, which can be easily re-targeted to any modern computer. Many targets are already supported by the standard GCC distribution, which should be checked before considering retargeting work.

Re-targeting requires considerable compiler expertise, appropriate host and target hardware, and a suitable compiler development system.

Please contact XGC Technology for a quotation.

---

## 7.2. Rehosting

The preferred host operating system is UNIX. This is because UNIX includes as standard, many of the utility programs that are required to make and install M68K Ada, and which are useful to operate M68K Ada. However M68K Ada may also be re-hosted (with reduced functionality) any version of Microsoft Windows that supports 32-bit programs.

---

### 7.2.1. Availability of Source Code

The complete source code for M68K Ada is included on the CD-ROM distribution.

---

### 7.2.2. Source Language

The Ada front end and the Ada predefined library are written in Ada 95. The C compiler (which is always included), the object code utilities, the debugger and the C libraries are written in ANSI C. The run-time start file, `art0.s`, is written in assembly language. Other standard UNIX languages (such as `make`, `YACC` and `Perl`) are used in the construction of the compiler.

---

### 7.2.3. System Dependencies

M68K Ada is designed to operate in a UNIX environment. This is not necessarily a UNIX system, but one that provides a POSIX compliant programming interface. Platforms such as Microsoft Windows may also be used but with reduced functionality.

---

M68K Ada is copyrighted commercial non-proprietary software.

The M68K Ada compiler and associated toolset are based on software from the *Free Software Foundation, Cambridge, MA*, and are supplied under their license. The M68K Ada run-time system and libraries are supplied under a special library license.

---

### *8.1. The Compiler License*

M68K Ada 95 is supplied under the *General Public License*, which is included on the CD-ROM.

This license requires us to make the source code available so that users are not prohibited from making further modifications.

Ready-to-install binary versions of the compiler, that have been thoroughly tested, are available for a fee.

The terms and conditions of the license permit you to copy the source or the binary versions, and to pass these to a third party, providing you do this on the same terms and condition under which the source or binary versions were supplied to you.

---

## 8.2. *The Run-Time License*

The run-time system and other run-time code are supplied on a license that follows the General Public License, but which explicitly allows you to use the source or object code in your application software without any of the GPL terms and conditions flowing down.

As a special exception, if other files instantiate generics from this unit, or you link this unit with other files to produce an executable, this unit does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU Public License.

The run-time license is supplied free of charge, and there are no recurring costs associated with using the run-time system.

---

## 8.3. *Support*

The medium on which M68K Ada is shipped, and the printed documentation, are warranted for six months from the time of shipment. They will be replaced free of charge if defective in any way.

The software is supplied with six months warranty, which may be extended for additional periods of 12 months, and applies to one project of up to six people at one site. The service offered includes regular product updates, advice on working around problems and general assistance with using the toolset or run-time system.

The warranty does not include training or customization. These are available for an additional fee.





---

We regularly test the XGC compilers against the ACVC test suite, and against its successor, the ACATS tests. While both of these are intended for compilers that no built-in restrictions, they offer good coverage of the Verison 1.7 compilers.

We have identified 3487 tests from ACATS Version 2.5 that are applicable to restricted compilers. Table 9.1, “The Validation Test Classes” [57] lists the number of tests in each section, and how many of those tests the compiler passes.

**Table 9.1. The Validation Test Classes**

<b>Group</b>	<b>Description</b>	<b>Number of Tests</b>	<b>Number of Applicable Tests</b>	<b>Number of Passes</b>
A	Class A tests check for acceptance (compilation) of language constructs that are	75	61	61

Group	Description	Number of Tests	Number of Applicable Tests	Number of Passes
	expected to compile without error.			
B <sup>a</sup>	Class B tests check that illegal constructs are recognized and treated as fatal errors.	1510	1510	1510
C	Class C tests check that executable constructs are implemented correctly and produce expected results.	2307	1835	1562 <sup>b</sup>
D	Class D tests check that implementations perform exact arithmetic on large literal numbers.	4	4	4
E	Class E tests check for constructs that may require inspection to verify.	32	9	6
L <sup>c</sup>	Class L tests check that all library unit dependencies within a program are satisfied before the program can be bound and executed, that circularity among units is detected, or that pragmas that apply to an entire partition are correctly processed.	89	68	68

<sup>a</sup>B tests are expected to fail with compilation time errors. Ones that are not applicable due to restrictions may therefore fail for different reasons from the original intention of the test, but nevertheless fail to compile and are therefore treated as passes.

<sup>b</sup>In group C, 303 tests did not print PASSED but terminated with an unhandled exception. In all 303 cases the exception was correctly raised then not handled because of the restriction on exception propagation.

<sup>c</sup>L tests are expected to give errors at compile time, bind time or link time and not to run.

# *Examples of Generated Code*

---

In this chapter we present examples of code generated by the Version 1.7 compiler.

---

## *A.1. The Sieve of Eratosthenes*

Compiler writers use the Sieve of Eratosthenes benchmark to check code quality and to compare run-time performance among compilers, languages and computers.

The benchmark uses the sieve method to compute the number of odd primes between 3 and 16383.

### **Example A.1. Source Code for Sieve**

```
procedure Sieve_Benchmark (Result : out Integer) is
  Size : constant := 8190;
  k, Prime : Natural;
  Count : Integer;
```

```
type Ftype is array (0 .. Size) of Boolean;
Flags : Ftype;
begin
  for Iter in 1 .. 10 loop
    Count := 0;

    for i in 0 .. Size loop
      Flags (i) := True;
    end loop;

    for i in 0 .. Size loop
      if Flags (i) then
        Prime := i + i + 3;
        k := i + Prime;
        while k <= Size loop
          Flags (k) := False;
          k := k + Prime;
        end loop;
        Count := Count + 1;
      end if;
    end loop;
  end loop;

  Result := Count;
end Sieve_Benchmark;
```

The generated code is given in Example A.2, “Generated Code for Sieve” [60]. The code was generated at optimization level 2 with all checks included.

### **Example A.2. Generated Code for Sieve**

```
1           .file   "sieve.adb"
2           gcc2_compiled.:
3           __gnu_compiled_ada:
4           .text
5           .even
6           .globl _ada_sieve_benchmark
7           _ada_sieve_benchmark:
8 0000 4E56 0000           link.w   %a6,#0
9 0004 48E7 3C20           movm.l  #0x3c20,-(%sp)
```

---

## The Sieve of Eratosthenes

---

```
10 0008 BBCF          cmp.l   %sp,%a5
11 000a 6B02          bmi.b   .+4
12 000c 4E45          trap    #5
13 000e 41EF E000     lea    (-8192,%sp),%a0
14 0012 BBC8          cmp.l   %a0,%a5
15 0014 6B02          bmi.b   .+4
16 0016 4E45          trap    #5
17 0018 2E48          move.l  %a0,%sp
18 001a 244F          move.l  %sp,%a2
19 001c 7801          moveq.l #1,%d4
20                                .even
21                                .L5:
22 001e 4283          clr.l   %d3
23 0020 203C 0000     move.l  #8190,%d0
23      1FFE
24 0026 41F2 0800     lea    (%a2,%d0.1),%a0
25                                .even
26                                .L9:
27 002a 10BC 0001     move.b  #1,(%a0)
28 002e 5388          subq.l  #1,%a0
29 0030 51C8 FFF8     dbra   %d0,.L9
30 0034 4240          clr.w   %d0
31 0036 5380          subq.l  #1,%d0
32 0038 64F0          jbcc   .L9
33 003a 4282          clr.l   %d2
34 003c 224A          move.l  %a2,%a1
35                                .even
36                                .L15:
37 003e 4A19          tst.b   (%a1)+
38 0040 6728          jbeq   .L14
39 0042 2002          move.l  %d2,%d0
40 0044 D080          add.l   %d0,%d0
41 0046 5680          addq.l  #3,%d0
42 0048 2202          move.l  %d2,%d1
43 004a 6012          jbra   .L29
44                                .even
45                                .even
46                                .L19:
47 004c 0C81 0000     cmp.l   #8190,%d1
47      1FFE
48 0052 6204          jbhi   .L20
49 0054 2041          move.l  %d1,%a0
50 0056 6002          jbra   .L21
51                                .even
```

```
52                                .L20:
53 0058 50FC                       trapt
54                                .even
55                                .L21:
56 005a 4230 A800                   clr.b   (%a0,%a2.1)
57                                .L29:
58 005e D280                       add.l   %d0,%d1
59 0060 0C81 0000                   cmp.l   #8190,%d1
60 0066 6FE4                       jble   .L19
61 0068 5283                       addq.l  #1,%d3
62                                .L14:
63 006a 5282                       addq.l  #1,%d2
64 006c 0C82 0000                   cmp.l   #8190,%d2
65 0072 6FCA                       jble   .L15
66 0074 5284                       addq.l  #1,%d4
67 0076 7A0A                       moveq.l #10,%d5
68 0078 BA84                       cmp.l   %d4,%d5
69 007a 6CA2                       jbge   .L5
70 007c 2003                       move.l  %d3,%d0
71 007e 4CEE 043C                   movm.l  -20(%a6),#0x43c
71                                FFEC
72 0084 4E5E                       unlk   %a6
73 0086 4E75                       rts
```

---

## A.2. Ackermann's Function

Using an informal functional notation, Ackermann's function is defined as follows:

$$\begin{aligned}A(0, n) &= n+1 \\A(m, 0) &= A(m-1, 1) \\A(m, n) &= A(m-1, A(m, n-1))\end{aligned}$$

From the point of view of benchmarking, Ackermann's function is interesting because it consists almost entirely of subprogram calls, and nests the calls deeply if required. The number of calls and the degree of nesting is controlled using the two arguments.

We use  $A(3,6)$  as the benchmark. This gives us 172233 calls, with a nesting depth of 511.

### **Example A.3. Ada Source Code for Ackermann's Function**

```
function Ackermann (M, N : in Integer) return Integer is
begin
  if M = 0 then
    return N + 1;
  elsif N = 0 then
    return Ackermann (M - 1, 1);
  else
    return Ackermann (M - 1, Ackermann (M, N - 1));
  end if;
end Ackermann;
```

Ackermann's function provides two opportunities for tail recursion optimization, both of which are taken here. The two parameters are passed in register, and the calling procedure saves any live registers across a call.

The generated code is given in Example A.4, “Generated Code for Ackermann's Function” [63]. For this version of the summary the code was generated at optimization level 2.

### **Example A.4. Generated Code for Ackermann's Function**

```
1          .file   "ack.adb"
2          gcc2_compiled.:
3          __gnu_compiled_ada:
4          .text
5          .even
6          .globl _ada_ackermann
7          _ada_ackermann:
8 0000 4E56 0000          link.w   %a6,#0
9 0004 2F0A              move.l   %a2,-(%sp)
10 0006 226E 0008       move.l   8(%a6),%a1
11 000a 206E 000C       move.l   12(%a6),%a0
12 000e BBCF           cmp.l    %sp,%a5
13 0010 6B02           bmi.b   .+4
14 0012 4E45           trap    #5
```

```
15                                     .L5:
16 0014 4A89                          tst.l  %a1
17 0016 6606                          jbn   .L2
18 0018 2008                          move.l %a0,%d0
19 001a 5280                          addq.l #1,%d0
20 001c 6022                          jbra  .L6
21                                     .even
22                                     .L2:
23 001e 4A88                          tst.l  %a0
24 0020 6608                          jbn   .L4
25 0022 5389                          subq.l #1,%a1
26 0024 307C 0001                     move.w #1,%a0
27 0028 60EA                          jbra  .L5
28                                     .even
29                                     .L4:
30 002a 45E9 FFFF                     lea   (-1,%a1),%a2
31 002e 4868 FFFF                     pea   -1(%a0)
32 0032 2F09                          move.l %a1,-(%sp)
33 0034 4EBA FFCA                     jsr   _ada_ackermann
34 0038 224A                          move.l %a2,%a1
35 003a 2040                          move.l %d0,%a0
36 003c 508F                          addq.l #8,%sp
37 003e 60D4                          jbra  .L5
38                                     .even
39                                     .L6:
40 0040 246E FFFC                     move.l -4(%a6),%a2
41 0044 4E5E                          unlk  %a6
42 0046 4E75                          rts
```



---

This Appendix defines how the Ada 95 restrictions, accessible through the pragma `Restrictions`, are supported. Unsafe features such as run-time dispatching and heap management are not supported in the run-time system, so all the restrictions that are relevant for these features are set to `True` by default.

The following restrictions are built in. That is, they cannot be turned off and are exploited by the compiler to offer better-quality generated code than would otherwise be possible.

- `No_Abort_Statements`
- `No_Dispatch`
- `No_Local_Protected_Objects`
- `No_Requeue`
- `No_Task_Attributes`
- `No_Task_Hierarchy`

- No\_Terminate\_Alternatives

The implementation-defined pragma Profile may also be used to set and unset restrictions that correspond to a certain application area. The profiles supported are as follows:

**Table B.1. Supported Profiles**

Profile Name	Description
XGC	This is the default profile and offers the least restrictions.
Ravenscar	This allows a limited form of tasking that includes static tasks, protected objects, the delay until statement and interrupts.
Restricted_Run_Time	This severely restricts the use of non-deterministic language features (including tasking) and is suitable for general avionics applications.
No_Run_Time	This profile prohibits all calls to the predefined Ada library and is useful for safety-critical applications. Calls to the compiler support library are not restricted.

Table B.2, “Profiles and Restrictions” [66] gives the individual restrictions for each profile. Note that the built-in restrictions apply to all profiles.

**Table B.2. Profiles and Restrictions**

Restriction	<i>Ada 95 Reference Manual</i> Section	Default	Ravenscar	Restricted_Run_Time
Boolean_Entry_Barriers	XGC (Ravenscar)	False	True	True
Immediate_Reclamation	RM H.4(10)	False	False	False
No_Abort_Statements	RM D.7(5), H.4(3)	True	True	True
No_Access_Subprograms	RM H.4(17)	False	True	True
No_Allocators	RM H.4(7)	False	False	True
No_Asynchronous_Control	RM D.9(10)	False	True	True
No_Calendar	XGC	False	True	True

<b>Restriction</b>	<b><i>Ada 95 Reference Manual Section</i></b>	<b>Default</b>	<b>Ravenscar</b>	<b><u>Restricted Run Time</u></b>
No_Delay	RM H.4(21)	False	False	True
No_Dispatch	RM H.4(19)	True	True	True
No_Dynamic_Interrupts	XGC	True	True	True
No_Dynamic_Priorities	RM D.9(9)	False	True	True
No_Elaboration_Code	XGC	False	False	True
No_Entry_Calls_In_Elaboration_Code	XGC	False	True	True
No_Entry_Queue	XGC	True	True	True
No_Enumeration_Maps	XGC	False	False	True
No_Exception_Handlers	XGC	False	False	True
No_Exceptions	RM H.4(12)	False	False	False
No_Fixed_Point	RM H.4(15)	False	False	False
No_Floating_Point	RM H.4(14)	False	False	False
No_Implementation_Attributes	XGC	False	False	True
No_Implementation_Pragmas	XGC	False	False	True
No_Implementation_Restrictions	XGC	False	False	True
No_Implicit_Conditionals	XGC	False	False	True
No_Implicit_Heap_Allocations	RM D.8(8), H.4(3)	False	True	True
No_Implicit_Loops	XGC	False	False	False
No_IO	RM H.4(20)	False	True	True
No_Local_Allocators	RM H.4(8)	False	True	True
No_Local_Protected_Objects	XGC	True	True	True
No_Nested_Finalization	RM D.7(4)	True	True	True
No_Protected_Type_Allocators	XGC	True	True	True
No_Protected_Types	RM H.4(5)	False	False	True
No_Recursion	RM H.4(22)	False	True	True
No_Reentrancy	RM H.4(23)	False	False	False
No_Relative_Delay	XGC	False	True	True
No_Requeue	XGC	True	True	True
No_Select_Statements	XGC (Ravenscar)	False	True	True

<b>Restriction</b>	<b><i>Ada 95 Reference Manual Section</i></b>	<b>Default</b>	<b>Ravenscar</b>	<b><u>Restricted Run Time</u></b>
No_Standard_Storage_Pools	XGC	True	True	True
No_Streams	XGC	True	True	True
No_Task_Allocators	RM D.7(7)	False	True	True
No_Task_Attributes	XGC	True	True	True
No_Task_Hierarchy	RM D.7(3), H.4(3)	True	True	True
No_Task_Termination	XGC	True	True	True
No_Terminate_Alternatives	RM D.7(6)	True	True	True
No_Unchecked_Access	RM H.4(18)	False	True	True
No_Unchecked_Conversion	RM H.4(16)	False	False	True
No_Unchecked_Deallocation	RM H.4(9)	True	True	True
No_Wide_Characters	XGC	False	True	True
Static_Priorities	XGC	False	True	True
Static_Storage_Size	XGC	False	True	True

Table B.3, “Profiles and Numerical Restrictions” [68] gives the restrictions concerning numerical limits.

**Table B.3. Profiles and Numerical Restrictions**

<b>Restriction</b>	<b><i>Ada 95 Reference Manual Section</i></b>	<b>Default</b>	<b>Ravenscar</b>	<b><u>Restricted Run Time</u></b>
Max_Asynchronous_Select_Nesting	RM D.7(18), H.4(2)	0	0	0
Max_Protected_Entries	RM D.7(14)	1	1	1
Max_Select_Alternatives	RM D.7(12)	Undefined	0	0
Max_Storage_At_Blocking	RM D.7(17)	0	0	0
Max_Task_Entries	RM D.7(13), H.4(2)	Undefined	0	0
Max_Tasks	RM D.7(19), H.4(2)	Undefined	Undefined	Undefined
Max_Entry_Queue_Depth	Ravenscar specific	1	1	1

---

Violation of the restriction Max\_Entry\_Queue\_Depth is detected at run time and raises the predefined exception Program\_Error.



---

This appendix lists the units in the Ada 95 predefined library, and indicates whether a unit is supported or not. The answer “Yes” means the unit is supported in the default profile, and maybe in the other profiles. The answer “Restricted...” means the unit is not supported in any profile because of a built-in restriction.

**Table C.1. Predefined Library Units**

<b>Unit Name</b>	<b>Supported?</b>
Ada	Yes
Ada.Asynchronous_Task_Control	Yes
Ada.Calendar	Yes <sup>a b</sup>
Ada.Characters	Yes
Ada.Characters.Handling	Yes
Ada.Characters.Latin_1	Yes
Ada.Characters.Wide_Latin_1	Yes
Ada.Command_Line	Not applicable

<b>Unit Name</b>	<b>Supported?</b>
Ada.Decimal	Yes
Ada.Direct_IO	Yes <sup>b</sup>
Ada.Dynamic_Priorities	Yes
Ada.Exceptions	Yes
Ada.Finalization	Restricted No_Implicit_Heap_Allocations
Ada.Interrupts	Yes
Ada.Interrupts.Names	Yes
Ada.IO_Exceptions	Yes
Ada.Numerics	Yes
Ada.Numerics.Complex_Elementary_Functions	Yes
Ada.Numerics.Complex_Types	Yes
Ada.Numerics.Discrete_Random	Not applicable
Ada.Numerics.Elementary_Functions	Yes
Ada.Numerics.Float_Random	Not applicable
Ada.Numerics.Generic_Complex_Elementary_Functions	Yes
Ada.Numerics.Generic_Complex_Types	Yes
Ada.Numerics.Generic_Elementary_Functions	Yes
Ada.Real_Time	Yes
Ada.Sequential_IO	Yes <sup>b</sup>
Ada.Storage_IO	Yes
Ada.Streams	Restricted No_Dispatch
Ada.Streams.Stream_IO	Restricted No_Dispatch
Ada.Strings	Yes
Ada.Strings.Bounded	Yes
Ada.Strings.Fixed	Yes
Ada.Strings.Maps	Yes
Ada.Strings.Maps.Constants	Yes
Ada.Strings.Unbounded	Not available



---

<b>Unit Name</b>	<b>Supported?</b>
Ada.Strings.Wide_Bounded	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Fixed	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Maps	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Maps.Wide_Constants	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Unbounded	Restricted No_Implicit_Heap_Allocations
Ada.Synchronous_Task_Control	Yes
Ada.Tags	Restricted No_Dispatch
Ada.Task_Attributes	No
Ada.Task_Identification	Yes
Ada.Text_IO	Yes <sup>b</sup>
Ada.Text_IO.Complex_IO	Not applicable
Ada.Text_IO.Editing	Not applicable
Ada.Text_IO.Text_Streams	Not applicable
Ada.Unchecked_Conversion	Yes
Ada.Unchecked_Deallocation	Restricted No_Unchecked_Deallocation
Ada.Wide_Text_IO	Not applicable
Ada.Wide_Text_IO.Complex_IO	Not applicable
Ada.Wide_Text_IO.Editing	Not applicable
Ada.Wide_Text_IO.Text_Streams	Not applicable
Calendar	Yes <sup>a b</sup>
Direct_IO	Yes <sup>b</sup>
IO_Exceptions	Yes
Interfaces	Yes
Interfaces.C	Yes
Interfaces.C.Pointers	Yes

---

<b>Unit Name</b>	<b>Supported?</b>
Interfaces.C.Strings	Yes
Interfaces.COBOl	Not applicable
Interfaces.FORTRAN	Not applicable
Machine_Code	Yes
Sequential_IO	Yes <sup>b</sup>
System	Yes
System.Address_to_Access_Conversions	Yes
System.Machine_Code	Yes
System.RPC	Not available (depends on Ada.Streams)
System.Storage_Elements	Yes
System.Storage_Pools	Not available (depends on Ada.Finalization)
Text_IO	Yes
Unchecked_Conversion	Yes
Unchecked_Deallocation	Restricted No_Unchecked_Deallocation

<sup>a</sup>Restricted to POSIX date range, which is Jan 1, 1970 to Jan 19, 2038

<sup>b</sup>When supported by appropriate system calls