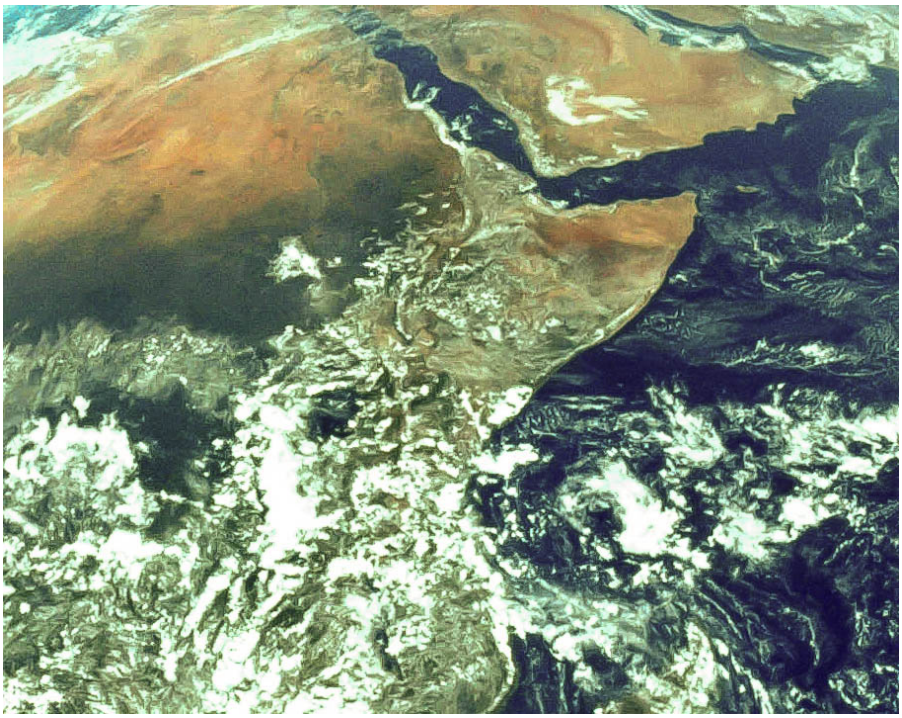
Leon Ada Technical Summary

For mission-critical applications using the Leon spacecraft computer



www.xgc.com

Leon Ada Technical Summary

For mission-critical applications using the Leon spacecraft computer

Order Number: LEON-ADA-TS-110101

XGC Technology

London
UK
<www.xgc.com>

Leon Ada Technical Summary: For mission-critical applications using the Leon spacecraft computer

by Chris Nettleton and Ellis Thomas

Published January 1, 2011

Abstract

This report presents technical and commercial information about Version 1.8 of the Leon Ada compilation system.

Acknowledgments

XGC Software 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 base-level C compiler, assembler and linker.

This manual is written in XML that conforms to DocBook Version 4.4. See *The DocBook web site* [http://www.docbook.org/] for more information.

Notice

Contents

Tables

Examples

Preface

This summary provides technical information about the Leon 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 1.8. Version 1.8 is the second version of Leon Ada to be released. Version 1.8 is based on ERC32 Ada Version 1.8. It includes changes required for the Leon architecture and other changes common across Version 1.8 products.

Chapter 1

Introduction

Leon 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 Leon Ada are as follows:

- Minimum program size less than 6K bytes, including the real-time kernel, and space for variables
- Accurate delays with 18 microseconds[1] average delay latency
- Low overhead 5K byte tasking system with 14 microseconds[1] task switch
- Full support for Ada interrupts attached to protected subprograms and fast interrupts with 3 microsecond latency
- Comprehensive printed and on-line user manuals
- Available off the shelf as a fully supported commercial product
- Evaluation copies available for down-load

[1] Simulated Leon at 100 MHz

- Built-in restrictions for mission-critical applications (see Appendix B, *Restrictions and Profiles* [59])
- Based on mature ERC32 Ada

1.1. Performance

Leon 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* [33].

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* [59].

1.3. User Documentation

The documentation provided with Leon Ada includes the following:

- *Getting Started with Leon 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 HTML format, and Adobe® PDF format.

1.4. Media

Leon Ada is shipped on CD-ROM, with on-line and printed user manuals, test report, Certificate of Conformance and Certificate of Origin.

1.5. Warranty

Leon Ada includes twelve months support to help users install and become familiar with the compiler and with using the Ada language on the Leon.

Chapter 2

Host and Target

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® Enterprise Linux (RHEL) Version 5 or above, or compatible Linux distribution, running on an IBM PC or compatible computer.
- Ubuntu® Linux Version 10.04 or above, or compatible Linux distribution, running on an IBM PC or compatible computer.

See Section 7.2, “Rehosting” [48] for information about additional host computers.

2.4. Target Configurations

For Leon Ada Version 1.8, the standard target is the Atmel AT697E. The generated program may be located in the Boot PROM, and copied to RAM for execution.

Other targets that are supported include the Atmel AT697F and the generic Leon 2 from The European Space Agency.

2.5. Target Operating System

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

2.6. Programming Support Environment

Leon 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 Leon 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.

Chapter 3

Language-Related Issues

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 Leon Ada and is organized according to the appropriate chapters and annexes of the Ada Manual.

3.1. Overview

Leon 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
- 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:

Feature	Ada RM Section
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 Leon 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 (See RM Section 3.9).

3.3.1. Uninitialized Variables

When the compile time options `-Wuninitialized -O` 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

Leon 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
`Short_Short_Integer`	8	-2^7	2^7-1
`Short_Integer`	16	-2^{15}	$2^{15}-1$
`Integer`	32	-2^{31}	$2^{31}-1$
`Long_Integer`	32	-2^{31}	$2^{31}-1$
`Long_Long_Integer`	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. Leon 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

Leon 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 the Leon.

Table 3.2, “Basic Attributes of Floating Point Types” [13] 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

Leon 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, 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
- the abort statement, and the asynchronous select statement
- the requeue statement

3.6. Section 6: Subprograms

Subprograms are fully supported.

3.7. Section 7: Packages

Except for finalization, 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” [15] gives the attributes of this type.

Table 3.3. Attributes of the Predefined Type Duration

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

Leon 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.

Leon 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 Leon 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 the operating system module (`art1.S`), 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 Leon Microprocessor, the entire program will consist of four items:

- The real-time kernel, `art0.S`, which contains the cold start entry point
- The startup module, `art1.S`, which contains the warm start 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 to provide basic services needed while a program is under development.

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

Leon 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” [18] specifies values from the predefined package System.

Table 3.4. Named Numbers from package System

Named Number	Value
Min_Int	-2^{63}
Max_Int	$2^{63} - 1$
Max_Binary_Modulus	2^{64}
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.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:

Feature	Reason for restriction
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. Preelaboration 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* [59].

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. The requested metrics are as follows: time for a call of Set = 120 clock cycles, time for a call of Read = 130 clock cycles.

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* [59].

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.

Chapter 4

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

Leon 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 Leon 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/leon-ada-1.8/`, for example, with the executable images in the directory `/opt/leon-ada-1.8/bin/`.

The main program of the Leon Ada compiler is called `leon-elf-gcc` and is located in the directory `/opt/leon-ada-1.8/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 leon-elf-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/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/adalib/.

Ada source files are always compiled in the context of a program library. While Leon 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 Leon 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 leon-elf-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$ leon-elf-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$ leon-elf-gcc -gnatlqv ackermann.adb
XGC Ada 1.8
```

```
Copyright 1992-2002 Free Software Foundation, Inc.

Compiling: ackermann_benchmark.adb (source file time stamp: 2008-07-27 03:31:04)

     1.    ------------------------
     2.    -- Ackermann_Benchmark --
     3.    ------------------------
     4.
     5.    function Ackermann_Benchmark (M, N : in Integer) return Integer is
     6.    begin
     7.       if M = 0 then
     8.          return N + 1;
     9.       elsif N = 0 then
    10.          return Ackermann_Benchmark (M - 1, 1);
    11.       else
    12.          return Ackermann_Benchmark (M - 1, Ackermann_Benchmark (M, N - 1));
    13.       end if;
    14.    end Ackermann_Benchmark;
    15.

 15 lines: No errors
```

The assembler can also generate a listing, as follows:

```
bash$ leon-elf-gcc -Wa,-a -c ackermann_benchmark.adb
SPARC GAS  /tmp/cc9JREJT.s                      page 1


   1                            .file  "ackermann_benchmark.adb"
   2                    gcc2_compiled.:
   3                    __gnu_compiled_ada:
   4                            .section        ".text._ada_ackermann_benchmark",#execinstr
   5                            .align  4
   6                            .global _ada_ackermann_benchmark
   7                            .proc   04
   8                    _ada_ackermann_benchmark:
   9 0000 9DE3BF98              save    %sp,-104,%sp
  10 0004 A0100018              mov     %i0,%l0
  11                    .L5:
  12 0008 80A42000              cmp     %l0,0
  13 000c 12800004              bne     .L2
  14 0010 80A66000              cmp     %i1,0
  15 0014 1080000C              b       .L6
  16 0018 B0066001              add     %i1,1,%i0
lots of output...
```

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

```
bash$ leon-elf-objdump -d ackermann_benchmark.o

ackermann_benchmark.o:     file format elf32-sparc


Disassembly of section .text._ada_ackermann_benchmark:

00000000 <_ada_ackermann_benchmark>:
   0:   9d e3 bf 98     save  %sp, -104, %sp
   4:   a0 10 00 18     mov  %i0, %l0
   8:   80 a4 20 00     cmp  %l0, 0
   c:   12 80 00 04     bne  1c <_ada_ackermann_benchmark+0x1c>
  10:   80 a6 60 00     cmp  %i1, 0
  14:   10 80 00 0c     b  44 <_ada_ackermann_benchmark+0x44>
  18:   b0 06 60 01     add  %i1, 1, %i0
  1c:   12 80 00 05     bne  30 <_ada_ackermann_benchmark+0x30>
...
```

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

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

`leon-elf-addr2line`
: which converts given target addresses to source file line numbers

`leon-elf-ar`
which is used to build object code libraries

`leon-elf-gdb`
which is the symbolic debugger

`leon-elf-gnatchop`
which may be used to divide a file that contains more than one compilation unit into one file for each unit

`leon-elf-gnatfind`
which is used to find Ada symbols in source files

`leon-elf-gnatls`
which is used to list Ada units

`leon-elf-gnatmake`
which uses the Ada rules to automatically compile, recompile and build an Ada program

`leon-elf-gnatprep`
which is an Ada pre-processor

`leon-elf-gnatpsta`
which prints the target package Standard

`leon-elf-gnatpsys`
which prints the target package System

`leon-elf-gnatxref`
which is the Ada cross reference tool

`leon-elf-nm`
which lists the symbols from object files

`leon-elf-objcopy`
which is used to copy and reformat object code files

`leon-elf-objdump`
which is used to dump information from object code files and includes an option to disassemble

`leon-elf-ranlib`
which generates an index to the contents of an archive and stores it in the archive

`leon-elf-run`
: which is the simulator

`leon-elf-sim`
: which is an interactive simulator

`leon-elf-size`
: which prints the size of an object code or executable file

`leon-elf-strings`
: which lists debug symbols and other strings in an object code file

`leon-elf-strip`
: which removes debug symbol table information from object code files

4.5. Debugging Facilities

The GNU debugger, gdb, as customized for Leon 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.

Leon Ada includes a target simulator that accurately simulates the target instruction set and timing. The simulator includes the following features:

- Simulates the entire Leon Microprocessor instruction set
- Simulates eight or 40 bit boot PROM, RAM, ERAM, IO memory, mass memory. Simulates EDAC and random SEUs.
- Simulates instruction timing, including wait states
- Prints statistics giving execution time, number of instructions in each class, number of nullified instructions
- Prints memory use report
- Prints task and interrupt trace reports

- Prints test coverage information, either for the whole program or for a given source file

Chapter 5

Performance and Capacity

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

5.1. Host Performance and Capacity

The compile time performance of Leon 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 Leon 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* [53].

The results of running the three benchmark programs *Sieve*, *Ackermann* and *Whetstone* are given in Table 5.1, “Benchmark Results” [34]. These programs were run on the simulator, with a 100 MHz clock, zero wait states on data read and write, and one wait state on instruction fetch.

Table 5.1. Benchmark Results

Benchmark	Result at 100 MHz	Percent stalls
Ackermann	127 mSec	5%
Sieve	25 mSec	3%
Whetstone	29636 KWIPS	24%

Table 5.2, “Task-Related Metrics” [35] gives timings for several task-related features. The clock frequency is 100 MHz.

Table 5.2. Task-Related Metrics

Metric	Clock Cycles	Time in Microseconds at 100 MHz
Interrupt latency (C.3.1 (15))	300	3
From call of trivial protected procedure to return from entry	1150	11
Call of Clock (D.8 (44))	13	1
Lateness of a delay (D.9 (13))	1100	11
Suspend_Until_True, where state is already True	50	1
Set_True to return from Suspend_Until_True	1010	10
Trivial protected procedure call (D.12 (6))	120	1

5.2.1. Optimization and Code Quality

Leon 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 = 84 bytes
- Instructions executed per call = 44. This includes instructions to handle window underflow and overlow on approximately 48 percent of the calls.

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 120 bytes are allocated for the task control block. In addition, there are 12 bytes for each task entry and the task's stack. 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 5500 bytes. The size of a minimal program that uses tasking (tasks, protected objects and delay statements) is approximately 10K bytes. These sizes include code, read-only data and variables, but exclude stack space.

Chapter 6

Cross-Compiler and Run-Time Interfacing

The internal structure of the Leon 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 Leon Ada compiler.

6.1.1. Background

The Leon 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 Leon Ada compiler a further driver program is supplied. This is called `leon-elf-gcc` and runs the compiler, assembler and linker targeted to the Leon 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 Leon 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 Leon 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:

- Executable Linkable Format (ELF)
- Common Object File Format (COFF)
- Motorola S-Records
- Intel Hex

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

6.3.3. Compiler Interfaces to Other Tools

Leon Ada provides information for other tools, notably the GNU debugger GDB and the GNU profiler GPROF. GPROF is not included with Leon 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. Leon 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

Leon Ada is shipped on CD-ROM. As with most UNIX software, installation is simple. For Solaris, Leon Ada is shipped as a Solaris package that is installed using the Solaris pkgadd command. For Linux Leon 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

Leon 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, exception management and low-level input/output. 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 SPARC standard and uses register windows with the save and restore instructions.

Register saving. Leon Ada uses the callee-save convention for saving registers across subprogram calls. On entry to a subprogram, the registers of the calling subprogram are saved by stepping the current register window pointer. Most of the time, there is sufficient space in the Leon register cache to complete the save without actually transferring any registers to memory. It is only when the cache fills that a transfer is needed. On return from a subprogram, the registers are restored using the restore instruction.

Parameter passing. Up to 6 words of parameters are passed in registers %o0 to %o5. Any further parameters are passed on the stack. For a function, or a procedure with a single `out` parameter, the result is passed out in register %o0.

The call instruction. Leon Ada uses the `call` 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[1] 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 Leon Microprocessor.

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

[1] A leaf subprogram is one that makes no subprogram calls.

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 packable array and record components to be allocated in adjacent bits without regard to byte boundaries.

6.6.4. Implementation of Ada Tasking

Leon 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 `art1.S`, and if required, `art0.S`.

6.7. Exception Handling System

Leon Ada supports exception declarations, the raise statement, and exception handlers. It does not support exception propagation. We expect Leon 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

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

Chapter 7

Re-targeting and Re-hosting

Leon 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 Leon 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 front end baseline, and run-time software written for Leon Ada.

7.1. Retargeting

Leon 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.

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 Leon Ada, and which are useful to operate Leon Ada. However Leon 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 Leon 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, `art1.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

Leon 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.

Chapter 8

Contractual Matters

Leon Ada is copyrighted commercial non-proprietary software.

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

8.1. The Compiler License

Leon 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 an 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 Leon Ada is shipped, and the printed documentation, are warranted for 12 months. They will be replaced free of charge if defective in any way.

The software is supplied with 12 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.

Chapter 9

Validation

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 1.8 compilers.

We have identified 3487 tests from ACATS Version 2.5 that are applicable to restricted compilers. Table 9.1, “The Validation Test Classes” [52] lists the number of tests in each section, and how many of those tests the compiler passes.

Table 9.1. The Validation Test Classes

Group	Description	Number of Tests	Number of Applicable Tests	Number of Passes
A	Class A tests check for acceptance (compilation) of language constructs that are expected to compile without error.	75	61	61
B [a]	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[b]
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 [c]	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

[a]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.
[b]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.
[c]L tests are expected to give errors at compile time, bind time or link time and not to run.

Appendix A

Examples of Generated Code

In this chapter we present examples of code generated by the Version 1.8 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 (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;
```

The generated code is given in Example A.2, “Generated Code for Sieve” [55]. The code was generated at optimization level 2 with checks suppressed.

Example A.2. Generated Code for Sieve

```
 1                          .file  "sieve_benchmark.adb"
 2                  gcc2_compiled.:
 3                  __gnu_compiled_ada:
 4                          .section ".text._ada_sieve_benchmark",#execinstr
 5                          .align  4
 6                          .global _ada_sieve_benchmark
 7                          .proc   04
 8                  _ada_sieve_benchmark:
 9 0000 033FFFF7            sethi   %hi(-8296),%g1
10 0004 82106398            or      %g1,%lo(-8296),%g1
11 0008 9DE38001            save    %sp,%g1,%sp
12 000c BA102001            mov     1,%i5
13 0010 05000007            sethi   %hi(8190),%g2
14 0014 B810A3FE            or      %g2,%lo(8190),%i4
15 0018 073FFFF8            sethi   %hi(-8192),%g3
16 001c 8407BFF8            add     %fp,-8,%g2
17 0020 B6008003            add     %g2,%g3,%i3
18 0024 8210001D            mov     %i5,%g1
19 0028 B0102000            mov     0,%i0
20                  .L32:
21 002c 84100018            mov     %i0,%g2
22 0030 C22EC002            stb     %g1,[%i3+%g2]
23                  .L27:
24 0034 8400A001            add     %g2,1,%g2
25 0038 80A0801C            cmp     %g2,%i4
26 003c 24BFFFFE            ble,a   .L27
27 0040 C22EC002            stb     %g1,[%i3+%g2]
28 0044 B2102000            mov     0,%i1
29 0048 C40EC019            ldub    [%i3+%i1],%g2
30                  .L31:
31 004c 80A0A000            cmp     %g2,0
32 0050 22800011            be,a    .L28
33 0054 B2066001            add     %i1,1,%i1
34 0058 852E6001            sll     %i1,1,%g2
35 005c B400A003            add     %g2,3,%i2
36 0060 8606401A            add     %i1,%i2,%g3
37 0064 80A0C01C            cmp     %g3,%i4
38 0068 3480000A            bg,a    .L29
39 006c B0062001            add     %i0,1,%i0
40 0070 05000007            sethi   %hi(8190),%g2
41 0074 8410A3FE            or      %g2,%lo(8190),%g2
42 0078 C02EC003            stb     %g0,[%i3+%g3]
43                  .L30:
44 007c 8600C01A            add     %g3,%i2,%g3
45 0080 80A0C002            cmp     %g3,%g2
46 0084 24BFFFFE            ble,a   .L30
```

```
 47 0088 C02EC003              stb     %g0,[%i3+%g3]
 48 008c B0062001              add     %i0,1,%i0
 49                    .L29:
 50 0090 B2066001              add     %i1,1,%i1
 51                    .L28:
 52 0094 80A6401C              cmp     %i1,%i4
 53 0098 24BFFFED              ble,a   .L31
 54 009c C40EC019              ldub    [%i3+%i1],%g2
 55 00a0 BA076001              add     %i5,1,%i5
 56 00a4 80A7600A              cmp     %i5,10
 57 00a8 24BFFFE1              ble,a   .L32
 58 00ac B0102000              mov     0,%i0
 59 00b0 81C7E008              ret
 60 00b4 81E80000              restore
```

A.2. Ackermann's Function

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

```
A(0, n) = n+1
A(m, 0) = A(m-1, 1)
A(m, n) = A(m-1, A(m, n-1))
```

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. 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 called procedure saves and restores registers using the register cache mechanism.

The generated code for the default optimization level is given in Example A.4, “Generated Code for Ackermann's Function” [57].

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

```
 1                      .file  "ackermann_benchmark.adb"
 2                  gcc2_compiled.:
 3                  __gnu_compiled_ada:
 4                      .section ".text._ada_ackermann_benchmark",#execinstr
 5                      .align  4
 6                      .global _ada_ackermann_benchmark
 7                      .proc   04
 8                  _ada_ackermann_benchmark:
 9 0000 9DE3BF98        save    %sp,-104,%sp
10 0004 A0100018        mov     %i0,%l0
11                  .L5:
12 0008 80A42000        cmp     %l0,0
13 000c 12800004        bne     .L2
14 0010 80A66000        cmp     %i1,0
15 0014 1080000C        b       .L6
16 0018 B0066001        add     %i1,1,%i0
17                  .L2:
18 001c 12800005        bne     .L4
19 0020 90100010        mov     %l0,%o0
20 0024 A0043FFF        add     %l0,-1,%l0
21 0028 10BFFFF8        b       .L5
22 002c B2102001        mov     1,%i1
23                  .L4:
24 0030 A0023FFF        add     %o0,-1,%l0
25 0034 40000000        call    _ada_ackermann_benchmark,0
26 0038 92067FFF        add     %i1,-1,%o1
27 003c 10BFFFF3        b       .L5
28 0040 B2100008        mov     %o0,%i1
29                  .L6:
30 0044 81C7E008        ret
31 0048 81E80000        restore
```

Appendix B

Restrictions and Profiles

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” [61] gives the individual restrictions for each profile. Note that the built-in restrictions apply to all profiles.

Table B.2. Profiles and Restrictions

Restriction	*Ada 95 Reference Manual* 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
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

Restriction	*Ada 95 Reference Manual* Section	Default	Ravenscar	Restricted_ Run_Time
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
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” [62] gives the restrictions concerning numerical limits.

Table B.3. Profiles and Numerical Restrictions

Restriction	*Ada 95 Reference Manual* Section	Default	Ravenscar	Restricted_ Run_Time
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.

Appendix C *The Predefined Library*

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

Unit Name	Supported?
Ada	Yes
Ada.Asynchronous_Task_Control	Yes
Ada.Calendar	Yes[ab]
Ada.Characters	Yes
Ada.Characters.Handling	Yes
Ada.Characters.Latin_1	Yes
Ada.Command_Line	Yes [c]
Ada.Decimal	Yes
Ada.Direct_IO	Yes[b]
Ada.Dynamic_Priorities	Yes
Ada.Exceptions	Yes
Ada.Finalization	Restricted No_Implicit_Heap_Allocations
Ada.Float_Text_IO	Yes
Ada.Float_Wide_Text_IO	No
Ada.Integer_Text_IO	Yes
Ada.Integer_Wide_Text_IO	No
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	Yes
Ada.Numerics.Elementary_Functions	Yes
Ada.Numerics.Float_Random	Yes
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[b]
Ada.Storage_IO	Yes

Unit Name	Supported?
Ada.Streams	Yes
Ada.Streams.Stream_IO	Restricted No_Streams, 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
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	Yes
Ada.Task_Identification	Yes
Ada.Text_IO	Yes[b]
Ada.Text_IO.Complex_IO	Yes
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[ab]
Direct_IO	Yes[b]

Unit Name	Supported?
IO_Exceptions	Yes
Interfaces	Yes
Interfaces.C	Yes
Interfaces.C.Pointers	Yes
Interfaces.C.Strings	Yes
Interfaces.COBOL	Not applicable
Interfaces.FORTRAN	Not applicable
Machine_Code	Yes
Sequential_IO	Yes[b]
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

[a]Restricted to POSIX date range, which is Jan 1, 1970 to Jan 19, 2038
[b]When supported by appropriate system calls
[c]With empty command line