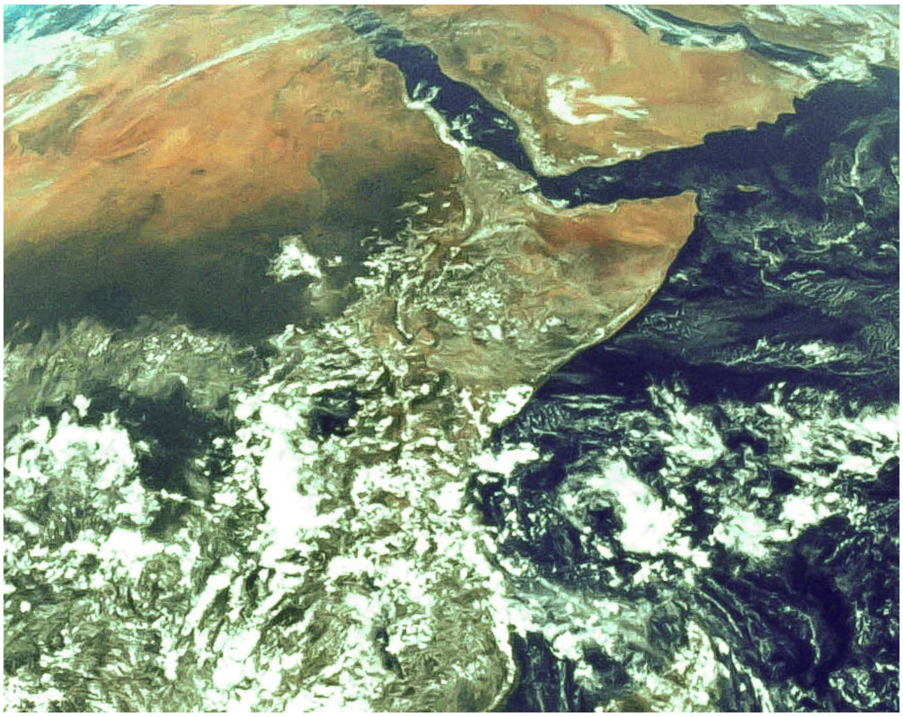
Getting Started with Leon Ada

Ada 95 Compilation System for Leon 2 (Atmel 697) Spacecraft Microprocessor



www.xgc.com

Getting Started with Leon Ada

Ada 95 Compilation System for Leon 2 (Atmel 697) Spacecraft Microprocessor

Order Number: LEON-ADA-GS-110101

XGC Technology

London
UK
Web: `<www.xgc.com>`

Getting Started with Leon Ada: Ada 95 Compilation System for Leon 2 (Atmel 697) Spacecraft Microprocessor

Publication date January 1, 2011

Acknowledgments

Leon Ada is based on Ada compiler technology developed by the GNAT team at New York University and includes software from the GNU C compiler, debugger and binary utilities developed by and on behalf of the Free Software Foundation, Inc., Cambridge, Massachusetts. Development of the mission-critical capability was funded by TRW Aerospace and the UK Ministry of Defence. Customization for the Leon was funded by XGC Technology.

Contents

About this Guide ***xi***

1 Audience **xi**
2 Related Documents **xi**
3 Reader's Comments **xii**
4 Documentation Conventions **xii**

Chapter 1 *Basic Techniques* ***1***

1.1 Hello World **1**
1.1.1 How to Prepare an Ada Program **2**
1.1.2 How to Compile an Ada Source File **2**
1.1.3 Using the gnatmake Command **3**
1.1.4 How to Run a Program on the XGC Simulator **3**
1.2 How to Recompile a Program **4**
1.3 The Generated Code **5**
1.4 What's in My Program? **8**
1.5 Restrictions **9**

Chapter 2 *Advanced Techniques* ***11***

2.1 How to Customize the File art0.S **12**
2.2 How to Customize the Linker Script File **14**

2.3 How to Get a Map File **14**
2.4 Generating PROM Programming Files **15**
2.5 Using the Debugger **17**
2.6 Using Optimizations **19**
2.7 Working with the Target **20**
2.8 Predefined Configurations **20**
2.9 Working with the EDAC **22**
2.10 System Calls **24**
2.10.1 How to Use Text_IO Without System Calls **25**
2.11 Improving Worst-Case Performance **27**

Chapter 3 *Real-Time Programs* ***29***

3.1 The Ravenscar Profile **30**
3.1.1 The Main Task **31**
3.1.2 Time Types **32**
3.1.3 Form of a Periodic Task **33**
3.1.4 Aperiodic Tasks **34**
3.2 Additional Predefined Packages **36**
3.3 Interrupts without Tasks **37**

Appendix A *Leon Compiler Options* ***39***

Appendix B *Leon Assembler Options and Directives* ***41***

B.1 Leon Options **41**
B.2 Enforcing aligned data **41**
B.3 Floating Point **42**
B.4 Leon Machine Directives **42**
B.5 Synthetic Instructions **43**

Appendix C *The Leon Simulator* ***47***

C.1 The Run Tool **49**
C.2 The Debugger **49**
C.3 The DCL Tool **49**
C.4 The Interactive Tool **49**

Appendix D *Using the Leon Debug Support Unit* ***51***

D.1 The Simulator in DSU Mode **52**
D.2 The Debugger and DSU **52**

Appendix E *Leon Simulator Run Mode* **53**

E.1 The Command Line **53**
E.2 Simulator Options **53**
E.3 Examples of Simulator Use **55**
E.4 Example of a Coverage Summary **57**

Appendix F *The package Ada.Interrupts.Names* **59**

Appendix G *The Host-Target Link* **63**

G.1 How to Change the UART Speed **64**
G.2 RS-232 Information **64**

Appendix H *Questions and Answers* **67**

Index **71**

Tables

B.1 Mapping of Synthetic Instructions to Leon Instructions **43**
F.1 Mapping of Interrupt Names to Priorities **60**
G.1 Pre-Computed Values for the UART Scaler **64**
G.2 Errors in bit rate **64**
G.3 Null Modem Wiring and Pin Connection **65**
G.4 The RS-232 Standard **65**

Examples

1.1 The Source File **2**
1.2 The Compile Command **2**
1.3 Binding and Linking **3**
1.4 Using gnatmake **3**
1.5 Running on the Simulator **3**
1.6 A Machine Code Listing **6**
1.7 Output from objdump **7**
1.8 Using the Size Command **7**
1.9 Using the Object Code Dump Program **8**
1.10 List of Included Object Code Files **9**
2.1 Creating a Custom art0.o **12**
2.2 Recompiling art0.S **12**
2.3 Rebuilding with a Custom art0.S **13**
2.4 Editing the Linker Script File **14**
2.5 Using a Custom Linker Script File **14**
2.6 How to Get a Map File **14**
2.7 The Map File **15**
2.8 Converting to Intel Hex **15**
2.9 Generating a HEX File **16**
2.10 Running an Intel Hex File **16**
2.11 Running an S-Record File **16**
2.12 Recompiling with the Debug Option **17**
2.13 Running under the Debugger **18**
2.14 Dump of Debug Information **18**
2.15 Running the simulator with the EDAC **24**
2.16 Code to Support Write **26**
2.17 Using Linker Sections **27**
3.1 Main Subprogram with Idle Loop **31**
3.2 Idle Loop with Power-Down **32**
3.3 A Periodic Task **33**
3.4 An Interrupt-Driven Task **35**
3.5 Example Interrupt Level Protected Object **38**

About this Guide

1. Audience

This guide is written for the experienced programmer who is already familiar with the Ada 95 programming language and with embedded systems programming in general. We assume some knowledge of the target computer architecture.

2. Related Documents

The *Leon Ada Technical Summary*, which summarises information about the toolset and the implementation-dependent features of the Ada 95 language.

The *XGC Ada User's Guide* describes the commands, options and scripts required to use the tool-set.

The *XGC Ada Reference Manual Supplement* documents the implementation-defined aspects of the Ada 95 programming language supported by the compiler.

The library functions, which are common to all XGC compilers, are documented in *The XGC Libraries*.

Documents on the AT697 and the SPARC V8 instruction set are available from Atmel Wireless and Microcontrollers (formerly Temic Semiconductors), http://www.atmel-wm.com/products/.

3. Reader's Comments

We welcome any comments and suggestions you have on this and other Leon Ada user manuals.

You can send your comments in the following ways:

- Internet electronic mail: `readers_comments@xgc.com`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number is printed on the title page of this manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of the software that you are using.

Technical support enquiries should be directed to the XGC Web Site [http://www.xgc.com/] or by email to support@xgc.com.

4. Documentation Conventions

This guide uses the following typographic conventions:

`%`, `$`
A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bash shell.

`#`
A number sign represents the superuser prompt.

`$ `**`vi hello.adb`**
Boldface type in interactive examples indicates typed user input.

file
: Italic or slanted type indicates variable values, place-holders, and function argument names.

[|], { | }
: In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

...
: In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated.

cat(1)
: A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the **cat** command in Section 1 of the reference pages.

Mb/s
: This symbol indicates megabits per second.

MB/s
: This symbol indicates megabytes per second.

Ctrl+**x**
: This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows. In examples, this key combination is printed in bold type (for example, **Ctrl**+**C**).

Chapter 1 Basic Techniques

To start with we'll write a small program and run it on the Leon simulator. This will give you a general idea of how things work. Later we will describe how to run a program on the real target computer.

1.1. Hello World

The subject of this section is a small program named “hello”. Using library functions and simulated input-output to do the printing, our program simply prints the line “`Hello World`” on the terminal. You will find the source code in the directory `examples` on the Leon Ada CD-ROM.

Three steps are needed to create an executable file from Ada source files:

1. The source file(s) must first be *compiled*.
2. The file(s) then must be *bound* using the Leon Ada binder.
3. All relevant object files must be *linked* to produce an executable file.

1.1.1. How to Prepare an Ada Program

Any text editor may be used to prepare an Ada program. If you use **emacs**, the optional Ada mode may be helpful in laying out the program. The program text is a normal text file. We will suppose that you have used your editor to prepare the following file:

Example 1.1. The Source File

```
with Ada.Text_IO;
procedure Hello is
begin
   Ada.Text_IO.Put_Line ("Hello World");
end Hello;
```

This file should be named `hello.adb`. [1]

1.1.2. How to Compile an Ada Source File

You can compile the file using the compile command in the following example:

Example 1.2. The Compile Command

```
$ leon-elf-gcc -c hello.adb
```

The command **leon-elf-gcc** is used to run the compiler. This command will accept files in several languages including Ada 95, C, assembly language and object code. It determines you have given it an Ada program by the filename extension (`.ads` or `.adb`), and will call the Ada compiler to compile the specified file.

The `-c` switch is always required when compiling an Ada source file. It tells **gcc** to stop after compilation. (For C programs, **gcc** can also do linking, but this capability is not used directly for Ada programs, so the `-c` switch must always be present.)

This compile command generates the file `hello.o` which is the object file corresponding to the source file `hello.adb`. It also generates a file `hello.ali`, which contains additional information used to check that an Ada program is consistent. To get an

[1] Leon Ada requires that each file contains a single compilation unit whose file name corresponds to the unit name with periods replaced by hyphens and whose extension is `.ads` for a spec and `.adb` for a body.

executable file, we then use **gnatbind** to bind the program and **gnatlink** to link the program.

Example 1.3. Binding and Linking

```
$ leon-elf-gnatbind hello.ali
$ leon-elf-gnatlink hello.ali
```

You may use the option `-v` to get more information about which version of the tool was used and which files were read.

1.1.3. Using the gnatmake Command

A better (simpler, quicker) method of carrying out these steps is to use the **gnatmake** command. **gnatmake** is a master program that invokes all of the required compilation, binding and linking tools in the correct order. In particular, it automatically recompiles any modified sources, or sources that depend on modified sources, so that a consistent compilation is ensured.

The following example shows how to use **gnatmake** to build the program `hello`.

Example 1.4. Using gnatmake

```
$ leon-elf-gnatmake hello
leon-elf-gcc -c hello.adb
leon-elf-gnatbind -x hello.ali
leon-elf-gnatlink hello.ali
$
```

The result is an executable file named `hello`.

1.1.4. How to Run a Program on the XGC Simulator

The program that we just built can be run on the simulator using the following command. If all has gone well, you will see the output line "Hello World".

Example 1.5. Running on the Simulator

```
$ leon-elf-run hello
Hello World
```

1.2. How to Recompile a Program

As you work on a program, you keep track of which units you modify and make sure you not only recompile these units, but also any units that depend on units you have modified.

gnatbind will warn you if you forget one of these compilation steps, so it is never possible to generate an inconsistent program as a result of forgetting to do a compilation, but it can be annoying to keep track of the dependencies. One approach would be to use a the UNIX make program, but the trouble with make files is that the dependencies may change as you change the program, and you must make sure that the make file is kept up to date manually, an error-prone process.

The Ada make tool, gnatmake takes care of these details automatically. In the following example we recompile and rebuild the example program, which has been updated.

```
$ leon-elf-gnatmake -v hello

GNATMAKE 1.8 Copyright 1995-2001 Free Software Foundation, Inc.
  -> "hello" final executable
  "hello.ali" being checked ...
  -> "hello.ali" missing.
leon-elf-gcc -c hello.adb
End of compilation
leon-elf-gnatbind -x hello.ali
leon-elf-gnatlink hello.ali
```

The argument is the file containing the main program or alternatively the name of the main unit. **gnatmake** examines the environment, automatically recompiles any files that need recompiling, and binds and links the resulting set of object files, generating the executable file, `hello`. In a large program, it can be extremely helpful to use **gnatmake**, because working out by hand what needs to be recompiled can be difficult.

Note that **gnatmake** takes into account all the intricate rules in Ada 95 for determining dependencies. These include paying attention to inlining dependencies and generic instantiation dependencies. Unlike some other Ada make tools, **gnatmake** does not rely on the dependencies that were found by the compiler on a previous compilation, which may possibly be wrong due to source changes. It works out the exact set of dependencies from scratch each time it is run.

The linker is configured so that there are defaults for the start file and the libraries `libgcc`, `libc` and `libada`. Other libraries, such as the standard C math library `libm`, are not included by default, and must be mentioned on the linker's command line.

1.3. The Generated Code

If you want to see the generated code, then use the compiler option `-Wa,-a`. The first part (`-Wa,`) means pass the second part (`-a`) to the assembler. To get a listing that includes interleaved source code, use the options `-g` and `-Wa,-ahld`. See *The Leon Ada Users Guide*, for more information on assembler options.

Here is an example where we generate a machine code listing.

Example 1.6. A Machine Code Listing

```
$ leon-elf-gcc -c -Wa,-a hello.adb
   1                            .file  "hello.adb"
   2                    gcc2_compiled.:
   3                    __gnu_compiled_ada:
   4                            .section .rodata
   5                            .align  8
   6                    .LC0:
   7 0000 48656C6C              .ascii "Hello World"
   7      6F20576F
   7      726C64
   8 000b 00                    .align  4
   9                    .LC1:
  10 000c 00000001              .long   1
  11 0010 0000000B              .long   11
  12 0014 00000000              .section ".text._ada_hello",#execinstr
  13                            .align  4
  14                            .global _ada_hello
  15                            .proc   020
  16                    _ada_hello:
  17 0000 9DE3BF90              save    %sp,-112,%sp
  18 0004 15000000              sethi   %hi(.LC0),%o2
  19 0008 9012A000              or      %o2,%lo(.LC0),%o0
  20 000c 15000000              sethi   %hi(.LC1),%o2
  21 0010 9212A000              or      %o2,%lo(.LC1),%o1
  22 0014 D03FBFF0              std     %o0,[%fp-16]
  23 0018 40000000              call    ada__text_io__put_line$2,0
  24 001c 9007BFF0              add     %fp,-16,%o0
  25 0020 81C7E008              ret
  26 0024 81E80000              restore
  27                            .size   _ada_hello, .-_ada_hello
...
```

You could also use the object code dump utility **leon-elf-objdump** to disassemble the generated code. If you compiled using the debug option `-g` then the disassembled instructions will be annotated with symbolic references.

Here is an example using the object code dump utility.

Example 1.7. Output from objdump

```
$ leon-elf-objdump -d hello.o

hello.o:     file format elf32-sparc


Disassembly of section .text._ada_hello:

00000000 <_ada_hello>:
   0:   9d e3 bf 90     save  %sp, -112, %sp
   4:   15 00 00 00     sethi  %hi(0), %o2
   8:   90 12 a0 00     mov  %o2, %o0   ! 0 <_ada_hello>
   c:   15 00 00 00     sethi  %hi(0), %o2
  10:   92 12 a0 00     mov  %o2, %o1   ! 0 <_ada_hello>
  14:   d0 3f bf f0     std  %o0, [ %fp + -16 ]
  18:   40 00 00 00     call  18 <_ada_hello+0x18>
  1c:   90 07 bf f0     add  %fp, -16, %o0
  20:   81 c7 e0 08     ret
  24:   81 e8 00 00     restore
```

You can see how big your program is using the **size** command. The sizes are in bytes. Note that the UNIX command `ls -s` gives you the size of the file rather than the size of the executable program.

Example 1.8. Using the Size Command

```
$ leon-elf-size hello.o
   text    data     bss     dec     hex filename
     64       0       0      64      40 hello.o
$ leon-elf-size hello
   text    data     bss     dec     hex filename
   7344     392     648    8384    20c0 hello
$
```

To get more detail you can use the object code dump program, and ask for headers.

Example 1.9. Using the Object Code Dump Program

```
$ leon-elf-objdump -h hello
hello:     file format elf32-sparc

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000016ec  40000000  40000000  00010000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text.adainit 00000028  400016ec  400016ec  000116ec  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.adafinal 00000008  40001714  40001714  00011714  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
$
```

1.4. What's in My Program?

You have written five lines of Ada, yet the size command says your program is over 7K bytes. What happened?

Answer: Your program has been linked with code from the Leon libraries. In addition to the application code, the executable program contains the following:

- The XGC kernel / real-time OS (art0)
- Program startup code (art1)
- Program elaboration code (adainit)
- Any Ada library packages mentioned in the context clauses of the source (libada)
- Any System packages needed by the compiler (also from libada)
- Object code from the compiler support library (libgcc)
- Object code from other libraries given on the linker command line (such as libm)

The following command will give you a list of the object files that have been linked into your program.

Example 1.10. List of Included Object Code Files

```
$ leon-elf-gnatmake hello.adb -largs -t
leon-elf-gnatbind -x hello.ali
leon-elf-gnatlink -t hello.ali
/opt/leon-ada-1.8/leon-elf/bin/ld: mode leon_sram
art0.o (/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/art0.o)
b~hello.o
./hello.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libada.a)a-textio.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libada.a)a-ioexce.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libgnat.a)x-except.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libgnat.a)x-malloc.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libc.a)_disable_preemption.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libc.a)_enable_preemption.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libc.a)_raise_constraint_error.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libc.a)_raise_exception.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libc.a)_raise_storage_error.o
(/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/libc.a)memcpy.o
...
```

In the example, the kernel (art0.o) accounts for most of the total size.

1.5. Restrictions

Before you go much further, you should be aware of the built-in restrictions. Leon Ada does not support the full Ada 95 language: it supports a restricted language that conforms to a formal *Profile* designed for high integrity applications.

The built-in restrictions prohibit the use of non-deterministic Ada features that would otherwise invalidate static program analysis. For a complete list of the default restrictions, see *The Leon Ada Reference Manual Supplement* or *The Leon Ada Technical Summary*.

Chapter 2

Advanced Techniques

Once you have mastered writing and running a small program, you'll want to check out some of the more advanced techniques required to write and run real application programs. In this chapter, we cover the following topics:

- Customizing the kernel (art0) and the linker script file
- Checking for stack overflow
- Generating PROM programming files
- Using the debugger
- Using optimizations
- Working on the target
- The boot PROM
- System calls
- The EDAC
- Improving Worst-Case Performance

2.1. How to Customize the File art0.S

On a real project you will almost certainly need to customize the kernel and the linker script file. These contain details of the target hardware configuration and project options such as running in user mode or supervisor mode.

The file `art0.S` contains instructions to initialize the arithmetic unit and floating point unit. The default file might be suitable for your requirements. The initial values of the system registers are defined in the file `register_inits.h`. You can see the source code in directory `/opt/leon-ada-1.8/leon-elf/src/kernel/`. If it is not suitable, make a copy in a working directory, then edit it as necessary.

Example 2.1. Creating a Custom art0.o

```
$ mkdir work
$ cd work
$ cp -a /opt/leon-ada-1.8/leon-elf/src/kernel/ .
$ vi kernel/config.h
$ vi kernel/register_inits.h
```

One of the configuration parameters you may wish to change is the clock speed. The default speed is 100 MHz. If your clock runs at (say) 35 MHz, then you should modify the statement in `config.h` that defines the clock frequency.

Once you have completed the changes, you must compile `art0.S` to generate an object code file named `art0.o`. This is the file that the default linker script will look for. Note that the compiler will select Atmel AT697E by default. If your target is a AT697F, then you should use the compile-time option `-mat697f`.

The following example gives the command you need:

Example 2.2. Recompiling art0.S

```
$ leon-elf-gcc -c kernel/art0.S
```

If you now rebuild your application program, the local file `art0.0` will be used in preference to the library file. In the following example we use the linker's -t option to list the files that are included in the link.

Example 2.3. Rebuilding with a Custom art0.S

```
$ leon-elf-gnatmake -f hello -largs -Wl,-t
leon-elf-gcc -c hello.adb
leon-elf-gnatbind -x hello.ali
leon-elf-gnatlink -Wl,-t hello.ali
/opt/leon-ada-1.8/leon-elf/bin/ld: mode leon_sram
art0.o
b~hello.o
./hello.o
... and more files ...
```

You can check that the system registers are initialized with the correct values by running your program on the simulator. For example:

```
$ leon-elf-sim hello
(sim) br main
Comment: DSU breakpoint 1 at 0x40001724
(sim) run
(sim) run
Comment: starting at 0x40000000

Comment: Hit breakpoint at PC 40001724
 <main>
(sim) show mem
Memory Configuration Register 1 (mcfg1) : 000000ff
-----------+----------------------------------+---------+----------------------
 iowdh      I/O bus width                      .        0  8 bits
 brdyn      Bus ready enable for I/O           .        0
 bexcn      Bus error enable                   .        0
 iows       I/O waitstates                     .        0  0
 iop        I/O protection                     .        0
 prbsz      Prom bank size                     .        0  8 Kbyte
 prwen      Prom write enable                  .        0
...
```

Note If you run a program built for 35 MHz on the simulator, be sure to specify a clock frequency of 35 MHz. The default is 100 MHz.

2.2. How to Customize the Linker Script File

The linker script file describes the layout of memory on the target computer and includes instructions on how the linker is to place object code modules in that memory. The default linker script file is `/opt/leon-ada-1.8/leon-elf/lib/ldscripts/leon_sram.x`. You should copy this file to your local directory, and edit as necessary.

Example 2.4. Editing the Linker Script File

```
$ cp /opt/leon-ada-1.8/leon-elf/lib/ldscripts/leon_sram.x myboard.ld
$ vi myboard.ld
```

You can then build a program using your custom linker script rather than the default, as follows:

Example 2.5. Using a Custom Linker Script File

```
$ leon-elf-gnatmake -f hello -largs -T myboard.ld
```

2.3. How to Get a Map File

If all you need is a link map, then you can ask the linker for one. This is a little more subtle than you may expect, because the option must be passed to the program **leon-elf-ld** rather than the Ada linker. Here is an example that generates a map named `hello.map`.

Example 2.6. How to Get a Map File

```
$ leon-elf-gnatmake hello -largs -Wl,-Map=hello.map
```

Example 2.7. The Map File

```
$ more hello.map
...
.init
 *(.init)

.text           0x40000000     0x16ec
                0x40000000               _sitext = .
                0x40000000               _stext = .
 *(.text)
 .text          0x40000000     0x16ec
/opt/leon-ada-1.8/lib/gcc-lib/leon-elf/2.8.1/art0.o
                0x40000000               __warm_start
                0x40000000               __trap_table
                0x40000a40               __reset_handler
                0x40000cfc               __window_overflow
...lots of output...
```

2.4. Generating PROM Programming Files

By default, the executable file is in Executable Linkable Format (ELF). Using the object code utility program **leon-elf-objcopy**, ELF files may be converted into several other industry-standard formats, such as Intel Hex, and Motorola S Records.

The following example shows how we convert a ELF file to Intel Hex format.

Example 2.8. Converting to Intel Hex

```
$ leon-elf-objcopy --output-target=ihex hello hello.ihex
```

If you don't need the ELF file, then you can get the linker to generate the Intel Hex file directly. Note that the Intel Hex file contains no debug information, so if you expect to use the debugger, you should generate the ELF file too.

Example 2.9. Generating a HEX File

```
$ leon-elf-gnatmake hello -largs -Wl,--oformat=ihex
$ more hello.ihex
:020000044000BA
:10000000108002900100000001000000010000000CB
:10001000A14800002910000381C52354A610201D0B
:10002000A14800002910000381C52354A610201BFD
:10003000A14800002910000381C52354A610201BED
:10004000A14800002910000381C52354A610201BDD
:10005000A14800002910000381C520FCA61020053E
:10006000A14800002910000381C52158A6102006D0
:10007000A14800002910000381C52354A610201DAB
:10008000A14800002910000381C52274A610200099
:10009000A14800002910000381C52354A610201D8B
...lots of output...
$
```

We can run the Intel Hex file, as in the following example:

Example 2.10. Running an Intel Hex File

```
$ leon-elf-run hello.ihex
Hello world
$
```

Or we can generate Motorola S Records, and run from there. Note that we use the option `-f` to force a rebuild.

Example 2.11. Running an S-Record File

```
$ leon-elf-gnatmake -f hello.adb -largs -Wl,--oformat=srec
$ more hello
S008000068656C6C6FE3
S31540000000108002900100000001000000010000000085
S31540000010A14800002910000381C52354A610201DC5
S31540000020A14800002910000381C52354A610201BB7
S31540000030A14800002910000381C52354A610201BA7
S31540000040A14800002910000381C52354A610201B97
S31540000050A14800002910000381C520FCA6102005F8
S31540000060A14800002910000381C52158A61020068A
S31540000070A14800002910000381C52354A610201D65
...lots of output...
$ leon-elf-run hello
Hello world
$
```

2.5. Using the Debugger

Before we can make full use of the debugger, we must recompile `hello.adb` using the compiler's debug option. This option tells the compiler to include information about the source code, and the mapping of source code to generated code. Then the debugger can operate at source code level rather than at machine code level.

The debug information does not alter the generated code in any way but it does make object code files much bigger. Normally this is not a problem, but if you wish to remove the debug information from a file, then use the object code utility **leon-elf-strip**.

This is how we recompile `hello.adb` with the `-g` option. There are other debug options too. See the *Leon Ada User's Guide* for more information on debug options.

Example 2.12. Recompiling with the Debug Option

```
$ leon-elf-gnatmake -f -g hello
leon-elf-gcc -c -g hello.adb
leon-elf-gnatbind -x hello.ali
leon-elf-gnatlink -g hello.ali
```

The debugger is **leon-elf-gdb**. By default the debugger will run a Leon program on the Leon simulator. If you prefer to run and debug on a real Leon then you must arrange for your target to communicate with the host using the debugger's remote debug protocol. This is described in Section 2.7, “Working with the Target” [20].

Example 2.13. Running under the Debugger

```
$ leon-elf-gdb hello
nettleto@cs1:~/xgc/src/leon-ada/examples$ leon-elf-gdb hello
GNU gdb (XGC leon-ada 1.8) 7.1
Copyright (C) 2010 Free Software Foundation, Inc.
This GDB was configured as "--host=i686-pc-linux-gnu --target=leon-elf".
For bug reporting instructions, please see:
<http://www.xgc.com/support/support.html>...
(gdb) br main
Breakpoint 1 at 0x40001728: file b~hello.adb, line 39.
(gdb)run
Starting program: .../examples/hello
Connected to the simulator.

Breakpoint 1, main () at b~hello.adb:39
34            adainit;
(gdb)c
Continuing.
Hello world

Program exited normally.
(gdb)quit
```

You can view the debug information using the object dump utility, as follows:

Example 2.14. Dump of Debug Information

```
bash$ leon-elf-objdump -G hello
hello:     file format elf32-sparc

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String

-1     HdrSym 0      847    00003c6c 1
0      SO     0      0      00000000 13     /home/nettleto/xgc/src/leon-ada/examples/
1      SO     0      0      00000000 1      b~hello.adb
2      LSYM   0      0      00000000 55     long int:t1=r1;-2147483648;2147483647;
3      LSYM   0      0      00000000 94     unsigned char:t2=@s8;r2;0;255;
...
```

2.6. Using Optimizations

Optimization makes your program smaller and faster. In most cases it also makes the generated code easier to understand. So think of the option `-O2` as the best setting, and only use other levels of optimization when you want to get something special.

Important We strongly recommend that you do not use any compiler options that change the generated code. Use the defaults.

The extent to which optimization makes a whole program smaller and faster depends on many things. In the case of `hello.adb` there will be little benefit since most of the code in the executable file is in the library functions, and these are already optimized.

The following example is more representative and shows the Whetstone benchmark program reduced to 49% of its size, and running nearly twice as fast. You can find Whetstone in the CD-ROM directory `benchmarks/`.

Here are the results when compiling with no optimization.

```
$ leon-elf-gcc -c -O0 whetstone.adb
$ leon-elf-size whetstone.o
   text    data     bss     dec     hex filename
   7752       0       0    7752    1e48 whetstone.o
$ leon-elf-gnatmake -f -O0 whetstone
$ leon-elf-run whetstone
,.,. Whetstone GTS Version 0.1
---- Floating point benchmark.
   - Whetstone Duration = 0.042511.
   - Whetstone Rating = 23523 KWIPS.
==== Whetstone  PASSED ===========================.
```

Here are the results when compiling with optimization level 2.

```
$ leon-elf-gcc -c -O2 whetstone.adb
$ leon-elf-size whetstone.o
   text    data     bss     dec     hex filename
   5272       0       0    5272    1498 whetstone.o
$ leon-elf-gnatmake -f -O2 whetstone
$ leon-elf-run whetstone
,.,. Whetstone GTS Version 0.1
```

```
---- Floating point benchmark.
   - Whetstone Duration = 0.033743.
   - Whetstone Rating = 29636 KWIPS.
==== Whetstone  PASSED ============================.
```

At optimization level 3, the compiler will automatically in-line calls of small functions. This may increase the size of the generated code, and the code will run faster. However the code motion due to inlining may make the generated code difficult to read and debug.

2.7. Working with the Target

Leon Ada also supports debugging on the target computer. Before you can do this, you must connect the target board to the host computer using two serial cables that include a *null modem*. One cable connects the board's serial connector A to the host, and is used to down-load the monitor and for application program input and output. The other cable connects to the board's serial connector B, and is used by the debugger to load programs, and to perform debugging operations.

##

2.8. Predefined Configurations

You may build your program to run from SRAM, or as a program that executes directly from PROM or as a program that is loaded from PROM and executes from RAM. These choices are known as linker emulations and are offered as predefined configurations.

You can ask the linker what emulations are supported, as follows:

```
$ leon-elf-ld -V
GNU ld (XGC leon-ada 1.8) 2.20
  Supported emulations:
   leon_sram
   leon_boot
   leon_sdram
   leon_prom
   leon_monitor
   leon_xgcmon
```

The emulation `leon_sram` (the default)
: The emulation `leon_sram`. All sections are located in SRAM starting at address `0x40000000`. The program's warm start entry point is `0x40000000`.

The emulation `leon_sdram`
: The emulation `leon_sdram`. All sections are located in SDRAM starting at address `0x60000000`. The program's warm start entry point is `0x60000000`.

The emulation `leon_boot`
: The emulation `leon_boot` builds a memory image that has an entry point at address `0x00000000`, and in which the program sections that contain instructions or data reside in the boot PROM. The start file includes additional code that copies these program sections into their proper locations in SRAM or SDRAM before branching to the entry point (at address `0x40000000`.

  This emulation is intended to replace the program `mkprom`. Unlike the output of `mkprom`, the memory image contains full debug information for the application program, which may be debugged on the simulator in the usual way.

The emulation `leon_prom`
: The emulation `leon_prom` builds a memory image that has an entry point at address `0x00000000` and with the data in SRAM.

The emulation `leon_monitor`
: The emulation `leon_monitor` builds a program that is suitable for downloading to the target computer and running with the XGC monitor. This means linking with the linker script file `xgcmon.ld` and using the start file `art1.S` in place of `art0.S`.

  Programs built with this emulation rely on the trap handlers in the monitor and all system calls are handled by the monitor.

To specifiy an emulation on the command line, use the linker option `-m` as follows:

```
$ leon-elf-gnatmake hello.adb -largs -Wl,-mleon_prom
```

This command will build an executable image that contains instructions starting at address zero that copy the main part of the image (i.e. the sections .text, .rodata and .idata) from an area of PROM into the main RAM starting at address 16#40000000#.

2.9. Working with the EDAC

Using memory that is 40 bits wide, the Leon's EDAC can correct single-bit errors in any 32-bit word, and can detect any double-bit error. The version 1.8 simulator contains a simulation of the EDAC and by default, this is switched off. In this state, the additional 8 bits of information are generated on each write to an EDAC-protected area of memory, but are not tested on memory reads.

The Leon offers EDAC protection for SRAM, for SDRAM and for the boot PROM, provided the PROM is the 40-bit wide kind. The 8-bit PROM has no EDAC protection, and no parity protection either. When EDAC simulation is switched on, then depending on further options set in the system registers, the EDAC may be used to correct and detect errors in memory. This raises several issues, some of which are very important from the point of view of running the odd test program.

- If you are simulating the 40-bit boot PROM, then the PROM must contain the extra 8 bits for each 32 bits of data. Normally you will use a tool to generate these bits and program them into the PROM or PROMs along with the rest of the data. To make life easier, the simulator's load function generates these extra bits for you. This means you can load a normal 32-bit wide program.

  Similarly if the simulator loads your program directly into SRAM or SDRAM, which is the case for prgorams built using the default options, it generates the checkbits. Of course, on the real target, this would happen without any special intervention because the loader would simply write your program to RAM with the EDAC generating the checkbits.

- For a data item that is 32 bits in size, the first memory write will write both the data and the extra check bits required by the EDAC. For byte size and half-word size items, the Leon will do a read-modify-write memory cycle with EDAC checking on the read and checkbit generation on the write. This means that byte size and half-word size items cannot be initialized individually; the initial memory write must be a whole word write otherwise the EDAC will report a failure when it reads the uninitialized memory word.

One solution to this is to initialize the whole of SRAM and SDRAM before the application program runs. This may take some time, possibly up to 20 seconds for a large memory.

In version 1.8, the cold-start emulation includes code that writes zero to each location in SRAM. The size of memory is known to art0 and this will need to be customized if your memory is not the same size. Also, if you wish to initialize SDRAM, you will need to add code to art0.

For application programs that do not include the cold start code, and this will be the majority of test programs, benchmarks and so on, we have the same EDAC problem. We therefore recommend running such programs with the EDAC switched off, and have set the simulator's EDAC option to off by default.

- If you wish to test your EDAC code, then one way to test at least part of it is to use the EDAC test feature offered by the Leon (and by the simulator). This allows you to specify the seven check bits on memory writes, and thereby introduce single-bit errors. You cannot introduce single-bit errors into the 32-bits of data because the EDAC write circuitary will always generate a correct parity bit and thwart any attemp to write data with bad parity (which indicates a single bit error).

  You can of course read the contents of a memory location and write it back with one or more of the bits changed. If you do this keeping the checkbits for the original contents then you will always get an uncorrctable error on the next read of that location.

  Version 1.8 also includes a generator for random single-bit errors that can be written to a random memory location at a given frequency, say one error per second, or maybe faster for times when you are keen to see you error recovery code working hard.

In the following example we run a program that requires the EDAC. We introduce random SEU errors into RAM at a rate of 10 per MB per second, which for a 8MB RAM is 80 per second.

Example 2.15. Running the simulator with the EDAC

```
$ leon-elf-run my_program -a "-mer 10 -trace-edac"
17280000: SEU before: addr 0035c5a0 bit 15, p 0, cb 00, data 00000000
17280000: ...  after:                       p 0, cb 00, data 01000000
34560000: SEU before: addr 00321e4c bit 31, p 0, cb 00, data 00000000
34560000: ...  after:                       p 0, cb 00, data 00000100
...
```

2.10. System Calls

A system call is the means by which application programs call an operating system. System calls are mostly used for input-output. The predefined Ada package `Ada.Text_IO` and the smaller package `XGC.Text_IO` map all input and output operations onto system calls, using open, close, read and write. The C language input-output functions declared in `<stdio.h>` use the same system calls.

For the convenience of the Ada programmer, XGC Ada includes the package `XGC.POSIX`, which declares the calls needed by `Ada.Text_IO` as Ada procedures each with an interface pragma. Note that the names of the procedures and the calls are the POSIX names, as used by most operating systems, and also by the XGC run-time system.

With XGC Ada, we have no operating system as such, just the kernel module art0. However, we support the system call mechanism using trap 80 (the SPARC standard) and when running on the simulator we map system calls to host system calls so that application programs can access host computer files during program development. This mapping can be disabled with a simulator option.

When running on the target, any system call will bring your program to an abnormal termination because the required system call handler is absent in the default configuration. The default system call handler is located in libc and supports an appropriate subset of calls. For example, read and write are directed to UART1 and may be used in a console dialog. You may wish to customise the default handler so that calls that would otherwise be non-operational could do something useful. For example, the call to get the time could be implemented to read the time from some external clock.

This can be done quite easily and an example system call handler is included with the source files in `/opt/leon-ada-1.8/leon-elf/src/libc/sys/schandler.c`. The handler is attached to the system call trap in the same fashion as other interrupts are attached to their handlers. In the example, a C function is provided to do the attaching.

2.10.1. How to Use Text_IO Without System Calls

Another way to support Text_IO is to replace the various system calls with calls to application code. For example, if all you need is the Put functionality in Text_IO, you can create your own version of `write` and have it do whatever you want. When your program is linked, the linker will use your version of `write` in place of the library version.

Example 2.16. Code to Support Write

```
--  UART registers. See AT697E, Tables 59 and 60

UAD1 : Unsigned_32;
for UAD1'Address use 16#80000070#;

UAS1 : Unsigned_32;
for UAS1'Address use 16#80000074#;

--  Protected object required to access system registers

protected UART1 is
   procedure Write (Ch : Character);
   pragma Interrupt_Handler (Write);
end UART1;

protected body UART1 is
   procedure Write (Ch : Character) is
   begin
      while (UAS1 and 16#00000004#) = 0 loop
         null;
      end loop;

      UAD1 := Character'Pos (Ch) and 16#ff#;
   end Write;
end UART1;

--  Export pragmas required for comatibility with C

procedure Write (
   Result : out Integer;
   Fd : in Natural;
   Buf : in System.Address;
   Count : in Natural);
pragma Export (C, Write, "write");
pragma Export_Valued_Procedure (Write, "write");

procedure Write (
   Result : out Integer;
   Fd : in Natural;
   Buf : in System.Address;
   Count : in Natural)
is
   Ada_Buf : String (1 .. Count);
   for Ada_Buf'Address use Buf;
begin
   for I in 1 .. Count loop
```

```
         UART1.Write (Ada_Buf (I));
      end loop;

      Result := Count;
   end Write;
```

2.11. Improving Worst-Case Performance

The instruction cache, the data cache, the register cache all tend to reduce the average time taken for a section of code to execute. However they offer little improvement in the worst-case time. One solution is to use a block of fast static RAM that has no cache and which completes memory cycles with minimum wait states.

Of course, this fast RAM is usually much smaller than required to the whole application, so it is necessary to partition the code and data into subprograms an objects that ought to go into fast ram, and the rest.

We can do this using the pragma `Linker_Section`. For example, to locate a subprogram in the linker section `.fastram`, while the rest of the code is located in the default section `.text`:

Example 2.17. Using Linker Sections

```
package Fast_RAM_Example is

   procedure P;
   pragma Linker_Section (P, ".fastram");

end Fast_RAM_Example;
```

Of course, you must customize the linker script file to specify where `.fastram` is located.

Chapter 3 *Real-Time Programs*

Leon Ada was designed for hard real-time applications that require accurate timing and a fast and predictable response to interrupts from peripheral devices. This is achieved with the following features:

- The Ravenscar profile
- The package `Ada.Real_Time` and a real-time clock with a resolution if one microsecond
- Preemptive priority scheduling with ceiling locking (5 microsecond task switch[1])
- Low interrupt latency (1 microseconds)
- The packages `Ada.Dynamic_Priorities`, `Ada.Synchronous_Task_Control` and `Ada.Task_Identification`
- Support for periodic tasks and task deadlines, as required by ARINC 653

Leon Ada also offers reduced program size by:

[1]Simulated AT697E at 100 MHz

- Optimized code generation
- Use of trap instructions to raise exceptions
- Smart linking where unused subprograms are eliminated
- Small run-time system size
- Optimizations that permit interrupt handling without tasking

This chapter describes how to use Ada tasks, and the associated language features, in an example real-time program.

3.1. The Ravenscar Profile

In support of safety-critcal applications, Ada 95 offers various restrictions that can be invoked by the programmer to prevent the use of language features that are known to be unsafe. Restrictions can be set individually, or can be set collectively in what is called a profile. XGC Ada supports all the Ada 95 restrictions and supports the implementation-defined pragma Profile. To get the compiler to work with the Ravenscar profile, you should place the following line at the top of each compilation unit.

```
pragma Profile (Ravenscar);
```

By default, Leon Ada supports a limited form of tasking that is a superset of what is supported by the Ravenscar profile. The built-in restrictions allow for statically declared tasks to communicate using protected types, the Ada 83 rendezvous or the predefined package `Ada.Synchronous_Task_Control`.

The Ravenscar profile prohibits the rendezvous and several other unsafe features. When using this profile, application programs are guaranteed to be deterministic and may be analyzed using static analysis tools.

The relevant Ada language features are as follows:

- The main task
- The pragma Priority
- Task specs and bodies

- Protected objects
- Interrupt handlers
- The delay until statement
- The package Ada.Real_Time

3.1.1. The Main Task

For a program that contains tasks, the main subprogram, which is at the root of the compilation unit graph, runs as task number 1. This is known as the main task. The TCB[2] for this task is declared in the run-time system, and its stack is the main stack declared in the linker script file. Other tasks are numbered from 2 in the order in which they are elaborated.

For other than a trivial program, the main task should probably be regarded as the idle task or background task. You can make sure that it runs at the lowest priority by the use of the pragma Priority in the declarative part of the main subprogram.

Example 3.1. Main Subprogram with Idle Loop

```
procedure T1 is
   pragma Priority (0);
begin
   loop
      null;
   end loop;
end T1;
```

You might want the background task to continuously run some built-in tests, or you may wish to switch the CPU into low-power mode until the next interrupt is raised.

Here is an example main subprogram that goes into low-power mode when there is nothing else to do. Note that the function `__xgc_power_down` is included in the standard library `libc`.

[2]Task Control Block, which holds the task state

Example 3.2. Idle Loop with Power-Down

```
pragma Profile (Ravenscar);

procedure T1 is
   pragma Priority (0);
   procedure Power_Down;
   pragma Import (C, Power_Down, "__xgc_power_down");
begin
   loop
      Power_Down;
   end loop;
end T1;
```

The rest of the program comprises periodic and aperiodic tasks that are declared in packages mentioned in the with list of the main subprogram.

Important In Leon Ada, there is no default idle task. If all of your application tasks become blocked, then the program will fail with Program_Error.

3.1.2. Time Types

The package `Ada.Real_Time` declares types and subprograms for use by real-time application programs. In Leon Ada, this package is implemented to offer maximum timing precision with minimum overhead.

The resolution of the time-related types is one microsecond. With a 32-bit word size, the range is approximately +/- 35 minutes. This is far greater than the maximum delay period likely to be needed in practice. For a 100 MHz processor, the lateness of a delay is approximately 10 microseconds. That means that given a delay statement that expires at time T, and given that the delayed task has a higher priority than any ready task, then the delayed task will restart at T + 10 microseconds. This lateness is independent of the duration of the delay, and represents the time for a context switch plus the overhead of executing the delay mechanism.

It is therefore possible to run tasks at quite high frequencies, without an excessive overhead. On a 100 MHz Leon, you can run a task at 1000Hz, with an overhead (in terms of CPU time) of approximately 2.5 percent, leaving 97.5 percent for the application program.

3.1.3. Form of a Periodic Task

The general form of a periodic task is given in the following example. You should note that tasks and protected objects must be declared in a library package, and not in a subprogram.

In this example, the task's three scheduling parameters are declared as constants, giving the example task a frequency of 100 Hz, and a phase lag of 3 milliseconds, and a priority of 3. You will have computed these parameters by hand, or using a third-party scheduling tool.

Example 3.3. A Periodic Task

```
package Periodic_Example is
   T0 : constant Time := Clock;
   --  Gets set at elaboration time

   Task1_Priority : constant System.Priority := 3;
   Task1_Period : constant Time_Span := To_Time_Span (0.010);
   Task1_Offset : constant Time_Span := To_Time_Span (0.003);

   task Task1 is
      pragma Priority (Task1_Priority);
   end Task1;
end Periodic_Example;

package body Periodic_Example is
   task body Task1 is
      Next_Time : Time := T0 + Task1_Offset;
   begin
      loop
         --  Do something
         Next_Time := Next_Time + Task1_Period;
         delay until Next_Time;
      end loop;
   end Task1;
end Periodic_Example;
```

The task must have an outer loop that runs for ever. The periodic running of the task is controlled by the delay statement, which gives the task a time slot defined by Offset, Period, and the execution time of the rest of the body.

The value of Task1_Period should be a whole number of microseconds, otherwise, through the accumulation of rounding errors, you may experience a gradual change in phase that may invalidate the scheduling analysis you did earlier.

3.1.4. Aperiodic Tasks

Like periodic tasks, aperiodic tasks have an outer loop and a single statement to invoke the task body.

In the following example, we declare a task that runs in response to an interrupt. You can use this code with a main subprogram to build a complete application that will run on the Leon simulator.

Here is the code for the package and its body:

Example 3.4. An Interrupt-Driven Task

```
package Aperiodic_Example is
   task Task2 is
      pragma Priority (1);
   end Task2;
end Aperiodic_Example;

with Ada.Interrupts.Names;
with Interfaces;
with Text_IO;

package body Aperiodic_Example is
   use Ada.Interrupts.Names;
   use Interfaces;
   use Text_IO;

   protected IO is
      procedure Handler;
      pragma Attach_Handler (Handler, UART_1_RX_TX);
      entry Get (C : out Character);
   private
      Rx_Ready : Boolean := False;
   end IO;

   protected body IO is
      procedure Handler is
         UAS1 : Unsigned_32;
         for UAS1'Address use 16#80000074#;
      begin
         Rx_Ready := (UAS1 and 16#00000001#) /= 0;
      end Handler;

      entry Get (C : out Character) when Rx_Ready is
         UAD1 : Unsigned_32;
         for UAD1'Address use 16#80000070#;
      begin
         C := Character'Val (UAD1 and 16#0000007f#);
         Rx_Ready := False;
      end Get;
   end IO;

   task body Task2 is
      C : Character;
   begin
      loop
        IO.Get (C);
```

```
         --  Do something with the character
         Put ("C = '"); Put (C); Put (''');
         New_Line;

       end loop;
    end Task2;

end Aperiodic_Example;
```

Points to note are as follows:

- The package Ada.Interrupts.Names declares the names of the 15 Leon interrupts. Note the associated priorities are listed in Table F.1, “Mapping of Interrupt Names to Priorities” [60].
- We use address clauses to declare memory-mapped IO locations.
- The type Unsigned_32 permits bitwise operators such as 'and' and 'or'.
- The interrupt handler runs in supervisor mode with the Processor Interrupt Level (PIL) set to the level of the interrupt.

3.2. Additional Predefined Packages

Programs that are not restricted to the Ravenscar Profile may also use the predefined packages `Ada.Asynchronous_Task_Control`, `Ada.Dynamic_Priorities`, `Ada.Synchronous_Task_Control` and `Ada.Task_Identification`.

The function `Current_Task` allows a task to get an identifier for itself. This identifier may then be used in calls the the subprograms in `Ada.Asynchronous_Task_Control`, which allow a task to be placed on hold, or to continue. Tasks that are on hold consume no CPU time but do retain their state.

The package `Ada.Task_Identification` allows a task to be aborted. In Leon Ada this places the task in a state from which it may be restarted using the subprograms in `XGC.Tasking.Stages`.

The base priority of any task (including the current task) may be requested or changed using the package `Ada.Dynamic_Priorities`. Note that if you change the priority of the current task within a protected operation then it is the base priority that changes: the active priority inherited from the protect object does not change. When the active priority is set back to the base priority is when the change takes effect.

3.3. Interrupts without Tasks

A protected operation that is attached to an interrupt must be a parameterless protected procedure. This is enforced by the pragma Attach_Handler and by the type Parameterless_Handler from package Ada.Interrupts. For interrupt handlers that have pragma Interrupt_Handler and are not attached to an interrupt is it convenient to allow both parameters and protected functions. The XGC compiler supports this as a legal extension to the Ada language.

In the special case where *all* the operations on a protected type are interrupt level operations, the XGC compiler will generate run-time system calls that avoid the use of the tasking system. Then only if tasks are required will the tasking system be present. This saves about 6K bytes of memory and reduces the amount of unreachable (and untestable) code.

Example 3.5. Example Interrupt Level Protected Object

```
with Ada.Interrupts.Names;

package body Example_Pack is
   use Ada.Interrupts.Names;

   protected UART_Handler is
      procedure Handler;
      pragma Attach_Handler (Handler, UART_1);
      --  Must be a parameterless procedure

      procedure Read (Buf : out String; Last : out Natural);
      pragma Interrupt_Handler (Read);
      --  Runs at interrupt level, may have parameters

      function Count return Integer;
      pragma Interrupt_Handler (Count);
      --  Runs at interrupt level
   end UART_Handler;

   protected body UART_Handler is
      ...
   end UART_Handler;

end Example_Pack;
```

Appendix A

Leon Compiler Options

These **-m** switches are supported on the Leon:

-mno-app-regs, **-mapp-regs**

Specify **-mapp-regs** to generate output using the global registers 2 through 4, which the SPARC SVR4 ABI reserves for applications. This is the default.

To be fully SVR4 ABI compliant at the cost of some performance loss, specify **-mno-app-regs**. You should compile libraries and system software with this option.

-mfpu, **-mhard-float**

Generate output containing floating-point instructions. This is the default.

-mno-fpu, **-msoft-float**

Generate output containing library calls for floating point.

-msoft-float changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option.

-mat697e

Select the Atmel AT697E (the default).

-mv7
Select a SPARC V7 chipset.

-mv8
Select a SPARC V8 chipset.

-mat697f
Select the Atmel AT697F.

-mcpu= *`cpu`*
Generate code for *`cpu`*, where *`cpu`* is either `at697e`, `v7`, `v8` or `at697f`. The default is `at697e`.

Appendix B Leon Assembler Options and Directives

B.1. Leon Options

-AAT697E
This is the default. It selects The Atmel AT697E.

-AV7
This selects standard SPARC V7.

-AV8
This selects standard SPARC V8.

-AAT697F
This selects standard Atmelo AT697F.

B.2. Enforcing aligned data

The assembler normally permits data to be misaligned. For example, it permits the **.long** directive to be used on a byte boundary. However, the native Sun-OS and Solaris assemblers issue an error when they see misaligned data.

You can use the **--enforce-aligned-data** option to make The assembler also issue an error about misaligned data, just as the Sun-OS and Solaris assemblers do.

The **--enforce-aligned-data** option is not the default because the compiler issues misaligned data directives when it initializes certain packed data structures (structures defined using the `packed` attribute). You may have to assemble with The assembler in order to initialize packed data structures in your own code.

B.3. Floating Point

The Leon uses IEEE floating-point numbers.

B.4. Leon Machine Directives

The assembler supports the following additional machine directives:

.common *`name`*, *`size`*, `"bss"`, *`alignment`*

> **.common** declares a named common area in the bss section. Normally the linker reserves memory addresses for it during linking, so no partial program defines the location of the symbol. Use **.common** to tell the linker that it must be at least *`length`* bytes long. The linker allocates space for each **.common** symbol that is at least as long as the longest **.common** request in any of the partial programs linked. *`length`* is an absolute expression.
>
> *`alignment`* gives the required linker alignment as a number of bytes.

.half

> This is functionally identical to **.short**.

.proc

> This directive is ignored. Any text following it on the same line is also ignored.

.reserve *`symbol`*, *`length`*, `".bss"`, *`alignment`*

> This must be followed by a symbol name, a positive number, and `"bss"` (or `".bss"`). This behaves somewhat like **.lcomm**, but the syntax is different.

`alignment` gives the required linker alignment as a number of bytes.

.seg

This must be followed by `"text"`, `"data"`, or `"data1"`. It behaves like **.text**, **.data**, or `.data 1`.

.skip

This is functionally identical to the **.space** directive.

.word

On the Leon, the **.word** directive produces 32 bit values, instead of the 16 bit values it produces on many other machines.

.xword

On the Leon processor, the **.xword** directive produces 64 bit values.

B.5. Synthetic Instructions

Table B.1, “Mapping of Synthetic Instructions to Leon Instructions” [43] describes the mapping of a set of synthetic (or "pseudo") instructions to actual Leon instructions. These synthetic instructions are provided by the Leon assembler for the convenience of assembly language programmers.

Note that synthetic instructions should not be confused with *pseudo-ops,*which typically provide information to the assembler but do not generate instructions. Synthetic instructions always generate instructions; they provide a more mnemonic syntax for standard Leon instructions.

The data in this table is based on Appendix A of *The SPARC Architecture Manual*, published by SPARC International, Inc.

Table B.1. Mapping of Synthetic Instructions to Leon Instructions

Synthetic Instruction	Leon Instruction(s)	Comment
cmp reg_{rs1},*reg_or_imm*	subcc reg_{rs1},*reg_or_imm*,%g0	*compare*

Synthetic Instruction	Leon Instruction(s)	Comment
jmp *address*	jmpl *address*,%g0	
call *address*	jmpl *address*,%o7	
tst reg_{rs2}	orcc %g0,reg_{rs2},%g0	*test*
ret	jmpl %i7+8,%g0	*return from subroutine*
retl	jmpl %o7+8,%g0	*return from leaf subroutine*
restore	restore %g0,%g0,%g0	*trivial* restore
save	save %g0,%g0,%g0	*trivial* save *(Warning: trivial save should only be used in kernel code!)*
set *value*,reg_{rd}	sethi %hi(*value*,reg_{rd}	*(when ((value & 0x1fff) == 0))*
	or	
	or %g0,*value*,reg_{rd}	*(when -4096 <= value <= 4095)*
	or	
	sethi %hi(*value*,reg_{rd};	*(otherwise)*
	or reg_{rd}, %lo(*value*),reg_{rd}	
not reg_{rs1},reg_{rd}	xnor reg_{rs1}, %g0,reg_{rd}	*one's complement*
not reg_{rd}	xnor reg_{rd}, %g0,reg_{rd}	*one's complement*
neg reg_{rs2},reg_{rd}	sub %g0,reg_{rs2}, reg_{rd}	*two's complement*
neg reg_{rd}	sub %g0,reg_{rd},reg_{rd}	*two's complement*
inc reg_{rd}	add reg_{rd}, 1,reg_{rd}	*increment by 1*
inc *const13*,reg_{rd}	add reg_{rd},*const13* ,reg_{rd}	*increment by const13*
inccc reg_{rd}	addcc reg_{rd}, 1,reg_{rd}	*increment by 1 and set icc*
inccc *const13*,reg_{rd}	addcc reg_{rd},*const13*,reg_{rd}	*increment by const13 and set icc*
dec reg_{rd}	sub *reg*, 1,reg_{rd}	*decrement by 1*
dec *const13*,reg_{rd}	sub *reg*,*const13*,reg_{rd}	*decrement by const13*

Synthetic Instruction	**Leon Instruction(s)**	**Comment**
deccc *reg*$_{rd}$	subcc *reg*, 1,*reg*$_{rd}$	*decrement by 1 and set icc*
deccc *const13*,*reg*$_{rd}$	subcc *reg*$_{rd}$,*const13*,*reg*$_{rd}$	*decrement by const13 and set icc*
btst *reg_or_imm*,*reg*$_{rs1}$	andcc *reg*$_{rs1}$,*reg_or_imm*,*%g0*	*bit test*
bset *reg_or_imm*,*reg*$_{rd}$	or *reg*$_{rd}$,*reg_or_imm*,*reg*$_{rd}$	*bit set*
bclr *reg_or_imm*,*reg*$_{rd}$	andn *reg*$_{rd}$,*reg_or_imm*,*reg*$_{rd}$	*bit clear*
btog *reg_or_imm*,*reg*$_{rd}$	xor *reg*$_{rd}$,*reg_or_imm*,*reg*$_{rd}$	*bit toggle*
clr *reg*$_{rd}$	or %g0,%g0,*reg*$_{rd}$	*clear(zero) register*
clrb *[address]*	stb %g0,*[address]*	*clear byte*
clrh *[address]*	sth %g0,*[address]*	*clear halfword*
clr *[address]*	st %g0,*[address]*	*clear word*
mov *reg_or_imm*,*reg*$_{rd}$	rd %g0,*reg_or_imm*,*reg*$_{rd}$	
mov %y,*reg*$_{rd}$	rd %y,*reg*$_{rd}$	
mov %asr*n*,*reg*$_{rd}$	rd %asr*n*,*reg*$_{rd}$	
mov %psr,*reg*$_{rd}$	rd %psr,*reg*$_{rd}$	
mov %wim,*reg*$_{rd}$	rd %wim,*reg*$_{rd}$	
mov %tbr,*reg*$_{rd}$	rd %tbr,*reg*$_{rd}$	
mov *reg_or_imm*,%y	wr %g0,*reg_or_imm*,%y	
mov *reg_or_imm*,%asr*n*	wr %g0,*reg_or_imm*,%asr*n*	
mov *reg_or_imm*,%psr	wr %g0,*reg_or_imm*,%psr	
mov *reg_or_imm*,%wim	wr %g0,*reg_or_imm*,%wim	
mov *reg_or_imm*,%tbr	wr %g0,*reg_or_imm*,%tbr	

Appendix C

The Leon Simulator

In the absense of a a real target computer equiped with some kind of debug interface, the only way to execute a Leon program is to use a Leon simulator.

The XGC Leon simulator runs on the host computer. It is written to be *demonstrably correct* and conforms to the relevant specifications. It simulates the SPARC V8 instructions set, with Atmel 697E errata, and with the peripheral registers and devices specified in the AT697 document.

The Leon IO functions using the Parallel IO (PIO) interface, the PCI Interface and the memory-mapped IO interface, are supported by shared libraries, which have a default skeleton version, and which can be replaced by custom versions.

The simulator is built into several tools, as follows:

- the run tool, **leon-elf-run**
- the debugger, **leon-elf-gdb**
- the DCL tool, **leon-elf-dcl**
- the interactive simulator, **leon-elf-sim**

The simulator is a near-complete implementation of the AT697E, and includes the following features:

- the Integer Unit (IU)
- the Floating Point Unit (FPU)
- the Debug Support Unit (DSU)
- the Debug Communications Link (DCL)
- the UARTS
- the interrupt mechanism
- the Parallel IO Interface (PIO)
- the PCI Interface (PCI)
- the Memory Interface
- PROM
- SRAM
- SDRAM
- the EDAC
- the Timers and Watch Dog

The following features are missing:

- the Instruction Cache
- the Data Cache
- the AHB Bus
- the JTAG unit
- the Register EDAC

C.1. The Run Tool

The run tool, **leon-elf-run**, executes the given program, with any UART output on the terminal, and using any custom IO sharable libraries.

C.2. The Debugger

The simulator is also built into the gdb debugger. Using the target 'sim' you can execure a program on the simulator while having a full set of debugger commands.

C.3. The DCL Tool

The DCL tool is a variant of the simulator that has a Debug Support Unit and DCL interface. The DCL tool can be connected to a serial RS232 interface, or to the TCP/IP network.

C.4. The Interactive Tool

Fianlly, the interactive tool, which is based on the Version 1.8 ERC32 Ada simulator **erc-elf-sim**.

Appendix D

Using the Leon Debug Support Unit

The Leon design includes a Debug Support Unit (DSU) that allows a debugger, such as the XGC debugger **leon-elf-gdb**, to control the Leon over a TCP/IP network or serial RS232 link. The functionality offered by the DSU is similar to what is offered by the debug monitor, but with the advantage that it is built into the hardware and therefore much more robust.

We have extended the set of targets supported by **leon-elf-gdb** with a new target that allows the debugger to link to the DSU.

We have also extended the Leon simulator so that it can be run in a mode where the DSU is active, and where the simulator can be controlled over a serial RS232 link.

Also we have extended the debugger and simulator so that the serial link may use TCP/IP over a network connection rather than the RS232 interface. While the hardware Leon does not support this, having such a connection during software development is of great benefit.

D.1. The Simulator in DSU Mode

The simulator is started using the command **leon-elf-dcl**. The command has one argument that is either the device name of the RS232 link, or the port number of the network link. The default is a network link using port 2000.

D.2. The Debugger and DSU

To connect the debugger to the Leon or simulator using the DSU, we use the following command:

```
(gdb) tar dsu /dev/ttyS0
```

To connect to the simulator via the network, use the following command:

```
(gdb) tar dsu localhost:2000
```

```
(gdb) tar dsu host.abc.com:2000
```

Appendix E

Leon Simulator Run Mode

E.1. The Command Line

The simulator command line has the form:

```
$ leon-elf-run switches files
```

E.2. Simulator Options

The simulator supports several options including the trace option (`-t`) and the statistics option (`-s`). Use the option `--help` for more information.

The first set of options are given to the `run` command along with the name of the execurtable file.

```
Options:
  -a "ARGS", --args "ARGS"    Pass ARGS to simulator
  -B,   --branch-report          Print branch coverage report
  -b,   --branch-summary         Print branch coverage summary
  -C,   --coverage-report        Print coverage report
  -c,   --coverage-summary      Print coverage summary
  -d T, --delay T               Delay trace for T uSec
  -e,   --call-time-report      Print calls CPU-time report (CPU time)
  -E,   --call-time-report2     Print calls real-time report (elapsed time)
  -f MOD, --file FILE           Report coverage for this source file only
  -h,   --help                  Print this message
  -I I, --interrupt I           Trigger trace on interrupt level I
  -i I, --pending I             Trigger trace on pending interrupt I
  -j,   --trace-traps           Trace traps
  -k,   --trap-time-report      Print trap time report (CPU time)
  -l T, --limit T               Time limit T uSec
  -m,   --trace-memory          Trace data memory cycles
  -M,   --trace-memory-wide     Trace data and instruction memory cycles
  -n,   --interrupt-report      Print interrupt report
  -N,   --interrupt-report-wide  Print interrupt report wide format
  -P PC, --pc PC                Trigger trace on pc = PC (use 0x for hex)
  -p,   --perf                  Print performance summary
  -r,   --ram-tags-report       Print RAM tags report with large blocks
  -R,   --RAM-tags-report       Print RAM tags report with small blocks
  -s,   --stats                 Print execution statistics
  -S,   --stop-on-fault         Stop simulation on fault trap
  -t,   --trace                 Trace instructions using 70 columns
  -T,   --trace-wide            Trace instructions using wide format
  -u U, --resolution U          Set task trace resolution to U uSec
  -V,   --verbose               Print additional information
  -v,   --version               Print version number
  -W,   --wider                 Widen a trace or report
  -w,   --wide                  Widen a trace or report
  -x,   --trace-calls           Trace subprogram calls
  -y,   --nosys                 Don't pass system calls to host
  -z,   --tasking-report        Print task switching report
  -Z,   --tasking-report-wide Print task switching report wide format
```

This second set of options are passed to the simulator within the run command, and to do this you need to use the run command's option -a as follows:

```
Simulator options are:
  -dsuen           Set the DSUEN pin
  -ta              Enable AHB bus back trace
  -ti              Enable instruction back trace
  -fill            Fill memory with test pattern
  -freq F          Set the clock frequency to F MHz (default 100)
  -mer R           Set memory error rate to R SEUs per second
  -pio BITS        PIO power-on bits (default 000)
  -rambs N         Set SRAM bank size to N (default 9, or 4 Mbytes)
  -sorbw            Stop on read-before-write
  -trace-edac       Trace any unusual EDAC operations
  -trace-events     Trace simulator's events
  -trace-interrupts Trcace interrupt requests and handling
  -trace-timers     Trace timer events
  -uart1 DEV        Connect UART1 to DEV (default stdin/stdout)
  -uart2 DEV        Connect UART2 to DEV
  -uben             Swap roles of UARTs 1 and 2
  -wdog             WDOG resets CPU
  -no-wdog          WDOG is ignored (default)
Trace format is:
     142.000 nzvc 15 spe 7 02000F34 sethi  %hi(0x2100000), %g1
           |    |  | ||| | |        |
           |    |  | ||| | |        - Disassembled instruction
           |    |  | ||| | - Program Counter (PC)
           |    |  | ||| - Current Window Pointer (CWP)
           |    |  | ||- Enable Traps (ET)
           |    |  | |- Previous Supervisor (PS)
           |    |  | - Supervisor mode (S)
           |    |  - Processor Interrupt Level (PIL)
           |    - IU Condition codes (ICC)
           - CPU time in microseconds

Report problems to <support@xgc.com>
```

The trace option prints each instruction as it is executed, along with the execution time in microseconds, and the instruction address. If the debug option was used when the source files were compiled, then source code line numbers will be printed too.

E.3. Examples of Simulator Use

The following example shows an instruction trace with line numbers. We have delayed the trace by 200 microseconds to skip to the lines of interest.

```
$ leon-elf-run -t -d 39 hello
-----------------------
-- Instruction trace --
-----------------------

------------+---------------+--------+---+-+--------+-------------------------------------
CPU time in  -(p)end-mask(e)- -----psr------         disassembled
```

```
microseconds fedcba9876543210 nzvc pil spe c       pc instruction
------------+----------------+--------+---+-+--------+---------------------------------------
      39.010        e    32           0   e 6 40004824 restore
main():
/home/nettleto/xgc/src/leon-ada/examples/b~hello.adb:57
 <main+18>
      39.020        e    32           0   e 7 400015D8 call  0x40001580
      39.030        e    32           0   e 7 400015DC nop
adainit():
/home/nettleto/xgc/src/leon-ada/examples/b~hello.adb:18
 <adainit>
      39.040        e    32           0   e 7 40001580 save  %sp, -104, %sp
/home/nettleto/xgc/src/leon-ada/examples/b~hello.adb:20
      39.050        e    32           0   e 6 40001584 sethi %hi(0x40100000), %o1
      39.060        e    32           0   e 6 40001588 mov  -1, %o0
      39.070        e    32           0   e 6 4000158C st  %o0, [ %o1 + 0x390 ]
/home/nettleto/xgc/src/leon-ada/examples/b~hello.adb:25
      39.100        e    32           0   e 6 40001590 call  0x40001644
      39.110        e    32           0   e 6 40001594 nop
ada__text_io___elabs():
/opt/leon-ada-1.8/leon-elf/src/libada/rts/a-textio.ads:208
 <ada__text_io___elabs>
      39.120        e    32           0   e 6 40001644 save  %sp, -176, %sp
/opt/leon-ada-1.8/leon-elf/src/libada/rts/a-textio.ads:214
      39.130        e    32           0   e 5 40001648 mov  1, %o1
      39.140        e    32           0   e 5 4000164C sethi %hi(0x40100400), %o0
...lots of output...
```

In this second example we run the demo program then interrupt with **Ctrl**+**C**. Using the `-z` option we get a short report on tasking showing task switches and the lock levels in the current task.

```
$ leon-elf-run demo -z
...
Ctrl-C
...


Tasks, y-axis is task number

  '>' Task running
  '.' Task in ready queue
  ' ' Task blocked

>   >  > >    > > >>   >>> > >  > > >  > > >>>>  >> > >    > > >>   >>|  1
```

```
                                                                      |  2
    >         >         >         >         >         >         >     |  3
    >         >         >         >         >         >         >     |  4
    >         >         >         >         >         >         >     |  5
    >         >         >         >         >         >         >     |  6
    >         >         >         >         >         >         >     |  7
    >    >    >    >    >    >    >    >    >    >    >    >    >    >|  8
    >    >    >    >    >    >    >    >    >    >    >    >    >    >|  9
    >    >    >    >    >    >    >    >    >    >    >    >    >    .| 10
    >    >    >    >    >    >    >    >    >    >    >    >    >    .| 11
    >    >    >    >    >    >    >    >    >    >    >    >    >    .| 12
>      > >      > >    > > >    >   >    > > >    > > >      > >    > | 13
>      > >      > >    > > >    >   >    > > >    > > >      > >    > | 14
>      > >      > >    > > >    >   >    > > >    > > >      > >    > | 15
>      > >      > >    > > >    >   >    > > >    > > >      > >    > | 16
>      > >      > >    > > >    >   >    > > >    > > >      > >    > | 17
                                                                      | 18
                                              >                       | 19
+---------+---------+---------+---------+---------+---------+---------0
-700000   -600000   -500000   -400000   -300000   -200000   -100000
                                                  uSec before end time


Locks and active priority for current task (above), y-axis is priority
Note: blank rows omitted

  'b' No locks, therefore base priority
  '1' One lock, active priority inherited

                                              1                       | 132
1   1  1 1    1 1 11   111 1 1  1 1 1  1 1 111   11 1 1    1 1 11   11| 127
b      b b      b b    b b b    b   b    b b b    b b b      b b    b |  11
    b    b    b    b    b    b    b    b    b    b    b    b    b    b|  10
                                              b                       |   5
b   b  b b    b b bb   bbb b b  b b b  b b bbbb  bb b b    b b bb   bb|   0
+---------+---------+---------+---------+---------+---------+---------0
-700000   -600000   -500000   -400000   -300000   -200000   -100000
                                                  uSec before end time
```

E.4. Example of a Coverage Summary

The following example shows a coverage summary for the program Hello.

```
$ erc-elf-run -c hello
Hello world

--------------------------
Execution Coverage Summary
--------------------------

 section section executable    fetched  percent section
 address    size      words      words coverage name
--------+-------+----------+----------+--------+---------------------
02000000    5748       1437        306       21 .text
02001674      40         10         10      100 .text.adainit
0200169c       8          2          2      100 .text.adafinal
020016a4       8          2          2      100 .text.__break_start
020016ac      48         12         12      100 .text.main
020016dc      44         11         11      100 .text._ada_hello
02001708      72         18         18      100 .text.xgc__exceptions___elabs
02001750     432        108        108      100 .text.ada__text_io___elabs
02001900      80         20         10       50 .text.ada__text_io__check_file_is_open
02001950      80         20         10       50 .text.ada__text_io__check_write_mode
020019a0      88         22         12       54 .text.ada__text_io__putc
020019f8     164         41         30       73 .text.ada__text_io__new_line
02001a9c     100         25         17       68 .text.ada__text_io__put
02001b00      84         21         21      100 .text.ada__text_io__put$3
02001b54      64         16         16      100 .text.ada__text_io__put_line
02001b94      52         13         13      100 .text.ada__text_io__put_line$2
02001bc8      28          7          0        0 .text.__xgc_raise_exception
02001be4      52         13         13      100 .text.memcpy
02001c18      76         19         14       73 .text.write
```

Appendix F

The package Ada.Interrupts.Names

The predefined package `Ada.Interrupts.Names` contains declarations for the Leon as follows:

```
package Ada.Interrupts.Names is

   --  Maskable asynchronous interrupts

   Internal_Bus_Error        : constant Interrupt_ID := 1;
   UART_2                    : constant Interrupt_ID := 2;
   UART_1                    : constant Interrupt_ID := 3;
   IO_Interrupt_0            : constant Interrupt_ID := 4;
   IO_Interrupt_1            : constant Interrupt_ID := 5;
   IO_Interrupt_2            : constant Interrupt_ID := 6;
   IO_Interrupt_3            : constant Interrupt_ID := 7;
   Timer_1                   : constant Interrupt_ID := 8;
   Timer_2                   : constant Interrupt_ID := 9;
   Unused_1                  : constant Interrupt_ID := 10;
   DSU_Trace_Buffer          : constant Interrupt_ID := 11;
   Unused_2                  : constant Interrupt_ID := 12;
   Unused_3                  : constant Interrupt_ID := 13;
   PCI                       : constant Interrupt_ID := 14;
   Unused_4                  : constant Interrupt_ID := 15;

   --  Unmaskable asynchronous interrupts
   Watchdog_Timeout          : constant Interrupt_ID := 15;

   --  Events. All reserved for the run-time system

   System_Call               : constant Interrupt_ID := 16;
   Breakpoint                : constant Interrupt_ID := 17;
   Suspend                   : constant Interrupt_ID := 18;
   Program_Exit              : constant Interrupt_ID := 19;
   Ada_Exception             : constant Interrupt_ID := 20;
   IO_Event                  : constant Interrupt_ID := 21;
   Timer_Interrupt           : constant Interrupt_ID := 22;
   Int_23                    : constant Interrupt_ID := 23;

   --  Faults. Available for application health management

   Deadline_Error            : constant Interrupt_ID := 24;
   Application_Error         : constant Interrupt_ID := 25;
   Numeric_Error             : constant Interrupt_ID := 26;
   Illegal_Request           : constant Interrupt_ID := 27;
   Stack_Overflow            : constant Interrupt_ID := 28;
   Memory_Violation          : constant Interrupt_ID := 29;
   Hardware_Fault            : constant Interrupt_ID := 30;
   Power_Fail                : constant Interrupt_ID := 31;

end Ada.Interrupts.Names;
```

The interrupt levels for the 15 interrupts are given in the following table:

Table F.1. Mapping of Interrupt Names to Priorities

Interrupt Name	Value of System.Interrupt_Priority
Internal_Bus_Error	129
UART_2	130
UART_1	131
IO_Interrupt_0	132
IO_Interrupt_1	133
IO_Interrupt_2	134
IO_Interrupt_3	135
Timer_1	136
Timer_2	137

Interrupt Name	**Value of System.Interrupt_Priority**
Unused_1	138
DSU_Trace_Buffer	139
Unused_2	140
Unused_3	141
PCI	142
Unused_4	143

Appendix G The Host-Target Link

The host-target link allows the debugger to communicate with the Leon debug support unit or with the monitor via UART1. The link uses an RS-232C interface connected to a serial port on the host computer, and connected to a compatible serial port on the target computer.

The RS232 standard applies to a connection between a computer and a modem. The standard does not apply to other kinds of connection, and for these there are many idiosyncrasies in the voltages, signals, pinout and connector types. For a host-to-target connection, the connecting cable must include a *null modem*. This is because both serial ports are configured to connect to a modem. The *null modem* is simply a cross over that wires the outputs from one port to the inputs of the other. Details of the wiring are given in Section G.2, “RS-232 Information” [64].

For the Leon, the RS232 link would be to connect the DCL link to the debugger and connect UART1 to a terminal, so that the output from Ada package Text_IO can be displayed.

The following information should help in setting up the links. You should not underestimate the effort required to get the links working and to recover when things go wrong.

G.1. How to Change the UART Speed

To change the speed of the two UARTs we must change the initial value of the UART scaler field in the system configuration register. This field occupies the most significant eight bits of register SYSCTR and its value is as follows.

Table G.1. Pre-Computed Values for the UART Scaler

Clock Frequency in MHz	**Bits per Second**							
	9600	**14400**	**19200**	**28800**	**38400**	**57600**	**76800**	**115200**
10	129	86	64	42	32	21	15	10
20	259	173	129	86	64	42	32	21
50	650	433	325	216	162	108	80	53
100	1301	867	650	433	325	216	162	108
200	2603	1735	1301	867	650	433	325	216

Because of rounding errors, the actual speed of the UART is usually different from the required speed. The following table gives the percentage error for each of the figures above.

Table G.2. Errors in bit rate

Clock Frequency in MHz	**Bits per Second**							
	9600	**14400**	**19200**	**28800**	**38400**	**57600**	**76800**	**115200**
10	1%	1%	2%	3%	2%	3%	9%	9%
20	1%	0%	1%	1%	2%	3%	2%	3%
50	0%	0%	0%	0%	0%	0%	2%	2%
100	0%	0%	0%	0%	0%	0%	0%	0%
200	0%	0%	0%	0%	0%	0%	0%	0%

G.2. RS-232 Information

The wiring of a null modem cable is given in Table G.3, “Null Modem Wiring and Pin Connection” [65].

Table G.3. Null Modem Wiring and Pin Connection

	25 Pin	9 Pin		9 Pin	25 Pin	
FG (Frame Ground)	1	N/A	----------	N/A	1	FG
TD (Transmit Data)	2	3	----------	2	3	RD
RD (Receive Data)	3	2	----------	3	2	TD
RTS (Request To Send)	4	7	----------	8	5	CTS
CTS (Clear To Send)	5	8	----------	7	4	RTS
SG (Signal Ground)	7	5	----------	5	7	SG
DSR (Data Set Ready)	6	6	----------	4	20	DTR
DTR (Data Terminal Ready)	20	4	----------	6	6	DSR

The RS-232 standard connection are given in Table G.4, “The RS-232 Standard” [65].

Table G.4. The RS-232 Standard

DB-25	DCE	DB-9			
1			AA	x	Protective Ground
2	TXD	3	BA	I	Transmitted Data
3	RXD	2	BB	O	Received Data
4	RTS	7	CA	I	Request To Send
5	CTS	8	CB	O	Clear To Send
6	DSR	6	CC	O	Data Set Ready
7	GND	5	AB	x	Signal Ground
8	CD	1	CF	O	Received Line Signal Detector
9			--	x	Reserved for data set testing
10			--	x	Reserved for data set testing
11				x	Unassigned
12	SCF			O	Secndry Rcvd Line Signl Detctr
13	SCB			O	Secondary Clear to Send
14	SBA			I	Secondary Transmitted Data

DB-25	**DCE**	**DB-9**			
15	DB			O	Transmisn Signl Elemnt Timng
16	SBB			O	Secondary Received Data
17	DD			O	Receiver Signal Element Timing
18				x	Unassigned
19	SCA			I	Secondary Request to Send
20	DTR	4	CD	I	Data Terminal Ready
21	CG			O	Signal Quality Detector
22		9	CE	O	Ring Indicator
23	CH/CI			I/O	Data Signal Rate Selector
24	DA			I	Transmit Signal Element Timing
25				x	Unassigned

Note DB-25 is the 25-pin connector.

Note DB-9 is the 9-pin connector, found on PCs.

Note Some SPARC Stations have a 25-pin connector with wiring for two RS232 interfaces (usually /dev/ttya and /dev/ttyb).

A spliiter cable is available from Sun.

Appendix H Questions and Answers

Here is a list of questions and answers.

Q: How do I change the installation directory? 67
Q: How do I un-install Leon Ada? 67
Q: Can I do mixed language programming? 68
Q: What is linked into my program over and above my Ada units? 68
Q: Can I build a program with separate code and data areas? 68
Q: Can I use the Leon Boot PROM? 68
Q: Which text editor should I use? 68
Q: Which UNIX shell should I use? 68
Q: Are programs restart-able? 69

Q: How do I change the installation directory?

A: On Solaris and Linux you can install the files in a directory of your choice then create a symbolic link from `/opt/leon-ada-1.8/` to that directory.

Q: How do I un-install Leon Ada?

A: On GNU/Linux, simply delete the directory `/opt/leon-ada-1.8/` and its contents.

On Solaris, you should use the pkgrm command. For example, Leon Ada Version 1.8 may be removed as follows:

```
# pkgrm XGClead17
```

Q: Can I do mixed language programming?

A: Yes. You can write a program using both `C` and `Ada 95` programming languages. In particular you can call the `C` libraries from code written in `Ada`.

Q: What is linked into my program over and above my Ada units?

A: When you build a program, the linker will include any run-time system modules that are necessary. The start file `art0.o` is always necessary. Other files such as object code for predefined Ada library units will be included only if they are referenced.

Q: Can I build a program with separate code and data areas?

A: Yes. Each object code module contains separate sections for instructions, read-only data, variable data and zeroized data. During the linking step, sections are collected together under the direction of the linker script file. The default is to collect each kind of section separately and to generate an executable file with separate code and data.

Q: Can I use the Leon Boot PROM?

A: Yes. The linker supports an emulation that locates the program in Boot PROM and which includes extra code to copy the application program from the Boot PROM into SRAM for execution.

Q: Which text editor should I use?

A: Leon Ada requires no special editing features and will work with your favorite text editor. If you use the emacs editor, then you will be able to run the compiler from the editor, and then relate any error messages to the source files. We recommend the universal UNIX editor vi.

Q: Which UNIX shell should I use?

A: We recommend the GNU Bash shell. It offers a much better user interface than other shells, and is kept up to date.

Q: Are programs restart-able?

A: Yes. The file `art0.S` contains code to initialize all variables in the `.data` section from a copy in read-only memory.

Index

Symbols
-AAT697E, 41
-AAT697F, 41
-AV7, 41
-AV8, 41
-enforce-aligned-data, 41
.common
  directive, 42

A
architectures
  Leon, 41
AT697E, 41
AT697F, 41

C
common
  directive, 42
common directive, Leon, 42

D
data alignment on Leon, 41
directive
  common, 42
  reserve, 42

F
floating point
  Leon, 42

H
half directive, Leon, 42

L
Leon
  architectures, 41
  data alignment, 41
  floating point, 42
  machine directives, 42
  options, 41
  support, 41
Leon options, 39

M
machine directives
  Leon, 42

O

options for Leon, 41

P

proc directive, Leon, 42

R

reserve directive, 42

S

seg directive, Leon, 43
skip directive, Leon, 43
SPARC V7, 41
SPARC V8, 41
symbol
    common, 42

W

word directive, Leon, 43

X

xword directive, Leon, 43