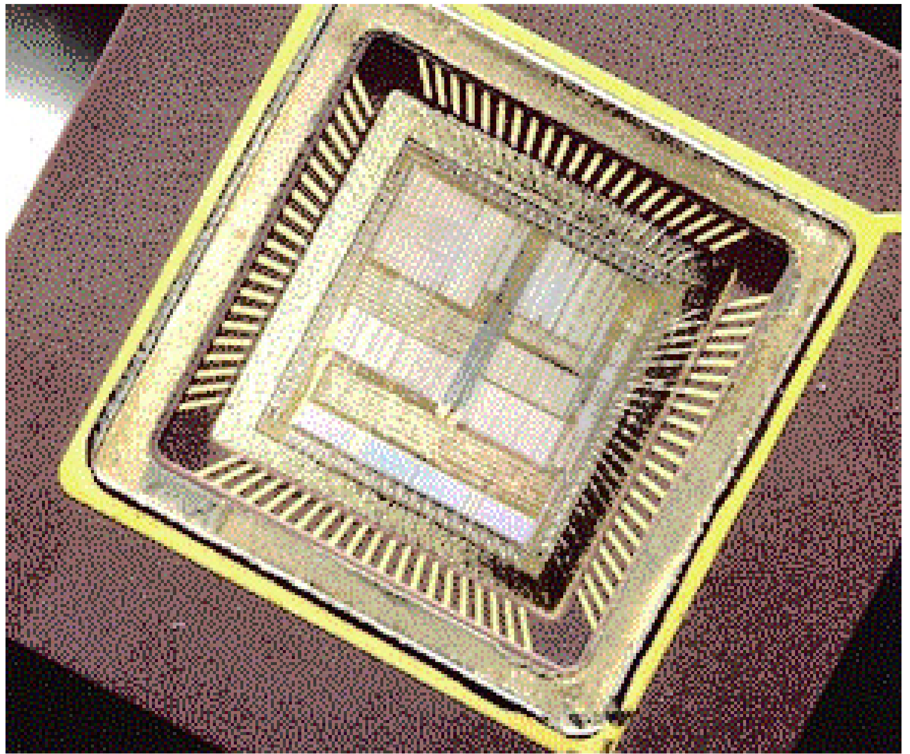
Getting Started with M1750 Ada

Ada 95 Compilation System for Spacecraft Microprocessors



www.xgc.com

Getting Started with M1750 Ada

Ada 95 Compilation System for Spacecraft Microprocessors

Order Number: M1750-ADA-GS-040730

XGC Technology

London
UK
Web: <www.xgc.com>

Getting Started with M1750 Ada: Ada 95 Compilation System for Spacecraft Microprocessors

Publication date July 30, 2004
© 1998, 1999, 2000, 2001, 2002, 2003, 2004 XGC Technology

Acknowledgments

M1750 Ada is based on GCC-1750, which was developed under contract with the European Space Agency, contract number 11935/NL/JG and on the front end of the GNAT Ada compiler developed at New York University. GCC-1750 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.

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 **2**
1.1.3 How to Run a Program on the Simulator **4**
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 Start File **12**
2.2 Using a Custom Linker Script File **13**
2.3 How to Get a Map File **13**
2.4 Generating PROM Programming Files **14**
2.5 Using the Debugger **15**
2.6 Using Optimizations **18**
2.7 Working with the Target **19**
2.7.1 How to Down-load the Debug Monitor **20**
2.7.2 Preparing a Program to Run under the Monitor **21**
2.8 Checking for Stack Overflow **22**
2.9 Expanded Memory **23**
2.10 System Calls **24**
2.10.1 How to Use Text_IO Without System Calls **25**

Chapter 3 *Real-Time Programs* **27**

3.1 The Ravenscar Profile **28**
3.1.1 The Main Task **29**
3.1.2 Periodic Tasks **30**
3.1.3 Form of a Periodic Task **31**
3.1.4 Aperiodic Tasks **32**
3.2 Additional Predefined Packages **35**
3.3 Interrupts without Tasks **36**

Appendix A *Expanded Memory* **39**

A.1 Expanded Memory Solutions **39**
A.1.1 The Single-Program Solution **40**
A.1.2 The Multi-Program Solution **40**

Appendix B *M1750 Compiler Options* **43**

Appendix C *M1750 Assembler Options and Directives* **45**

C.1 MIL-STD-1750 Options **45**
C.2 Floating Point **46**

C.3 M1750 Machine Directives **47**
C.4 Opcodes **48**
C.4.1 Extended Floating Load Register (ELFR) **48**
C.4.2 Expanded Memory Support **48**
C.4.3 Branch Improvement **49**
C.4.4 XIO Commands **49**
C.4.5 Special Characters **50**

Appendix D *M1750 Simulator Options* ***51***
D.1 The Command Line **51**
D.2 Command Line Switches **53**
D.3 Examples of Simulator Use **54**
D.3.1 Tracing Simulation **54**
D.3.2 Tasking Reports **56**
D.3.3 The RAM Tags Report **58**
D.4 How to Customize the Simulator **60**

Appendix E *The package Ada.Interrupts.Names* ***63***

Appendix F *The Host-Target Link* ***65***
F.1 RS-232 Information **65**

Appendix G *Questions and Answers* ***69***

Index ***73***

Tables

C.1 M1750 Pseudo Operations for Branches **49**
F.1 The RS-232 Standard **66**
F.2 Null Modem Wiring and Pin Connection **67**

Examples

1.1 The Source File **2**
1.2 The Compile Command **2**
1.3 Binding and Linking **3**
1.4 Using gnatmake to Compile **3**
1.5 Using gnatmake to Recompile **4**
1.6 Generating a Machine Code Listing **6**
1.7 Output from Object Code Dump Program **7**
1.8 Using the Size Program **7**
1.9 Object Code Section Headers **8**
2.1 Creating a Custom Start File **12**
2.2 Making a Custom Linker Script File **13**
2.3 Using the Custom Linker Script File **13**
2.4 The Map File **14**
2.5 Running under the Debugger **17**
2.6 Dump of Debug Information **18**
2.7 Remote Configuration File **20**
2.8 Remote Debugging **22**
2.9 Stack Overflow Check **23**
2.10 Code to Support Write **26**
3.1 Main Subprogram with Idle Loop **29**
3.2 Idle Loop with Power-Down **30**
3.3 A Periodic Task **32**
3.4 An Interrupt-Driven Task **34**
3.5 Example Interrupt Level Protected Object **37**
C.1 XIO Command in Ada **50**
D.1 Simulator Help **52**
D.2 Tracing Simulation **55**
D.3 Tracing Tasking **57**
D.4 A RAM Tags Report **59**

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 *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*.

The M1750 Ada Technical Summary, which contains technical and commercial information about the compiler.

3. Reader's Comments

We welcome any comments and suggestions you have on this and other XGC 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.c`**
: 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 instruction set 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 chapter is a small program called “hello”. Using library functions and simulated input-output to do the printing, it simply prints the message “`Hello World`” on the terminal. You will find the source code in the directory `/examples` on the M1750 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 M1750 Ada binder.

3. All appropriate object files must be *linked* to produce an executable file.

1.1.1. How to Prepare an Ada Program

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

Example 1.1. The Source File

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

The Ada compiler 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. This example file should be named `hello.adb`.

1.1.2. How to Compile

You can compile the file using the following command:

Example 1.2. The Compile Command

```
$ m1750-coff-gcc -c hello.adb
```

The command **m1750-coff-gcc** is used to run the compiler. This command will accept programs 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. 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 executable file, we then use **gnatbind** to bind the program and **gnatlink** to link the program.

Example 1.3. Binding and Linking

```
$ m1750-coff-gnatbind hello.ali
$ m1750-coff-gnatlink hello.ali
```

The result is an executable file called `hello`.

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

A simpler 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 to Compile

```
$ m1750-coff-gnatmake hello
m1750-coff-gcc -c hello.adb
m1750-coff-gnatbind -x hello.ali
m1750-coff-gnatlink hello.ali
```

Again, the result is an executable file called `hello`.

1.1.3. How to Run a Program on the 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 message "Hello World".

```
$ m1750-coff-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.

The binder, 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.

Example 1.5. Using gnatmake to Recompile

```
$ m1750-coff-gnatmake hello.adb
$ m1750-coff-gnatmake -v hello
GNATMAKE 1.7 Copyright 1995-2001 Free Software Foundation, Inc.
  "hello.ali" being checked ...
  -> "hello.adb" time stamp mismatch
m1750-coff-gcc -c hello.adb
End of compilation
m1750-coff-gnatbind -x hello.ali
m1750-coff-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 library `libgcc`, `libc` and `libada`. Other libraries, such as the standard C math library `libm.a`, 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 XGC Ada Users Guide*, for more information on assembler options.

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

Example 1.6. Generating a Machine Code Listing

```
$ m1750-coff-gcc -c -O2 -Wa,-a hello.adb
   1                                    .file  "hello.adb"
   2                            gcc2_compiled.:
   3                            __gnu_compiled_ada:
   4                                    .section .rdata,"r"
   5                            .LC0:
   6 0000 0048                          .word  72
   7 0002 0065                          .word  101
   8 0004 006C                          .word  108
   9 0006 006C                          .word  108
  10 0008 006F                          .word  111
  11 000a 0020                          .word  32
  12 000c 0057                          .word  87
  13 000e 006F                          .word  111
  14 0010 0072                          .word  114
  15 0012 006C                          .word  108
  16 0014 0064                          .word  100
  17                            .LC1:
  18 0016 0001                          .word  1
  19 0018 000B                          .word  11
  20                                    .text
  21                            .global _ada_hello
  22                            _ada_hello:
  23 0000 9FEE                          pshm   r14,r14
  24 0002 81EF                          lr     r14,r15
  25 0004 81BF                          lr     r11,r15
  26 0006 4AB9 8000                     xorm   r11,0x8000
  27 000a F0B0 0000                     c      r11,_stack_limit
  28 000e 7B02                          bge    .+4
  29 0010 7708                          bex    8
  30 0012 8500 0000                     lim    r0,.LC0
  31 0016 8510 000B                     lim    r1,.LC1
  32 001a 7EF0 0000                     sjs    r15,ada__text_io__put_line__2
  33 001e 81FE                          lr     r15,r14
  34 0020 8FEE                          popm   r14,r14
  35 0022 7FF0                          urs    r15
...
```

You could also use the object code dump utility **m1750-coff-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 Object Code Dump Program

```
$ m1750-coff-objdump -d hello.o

hello.o:     file format coff-m1750

Disassembly of section .text:

00000000 <_ada_hello>:
   0:   9f ee           pshm   r14,r14
   2:   81 ef           lr     r14,r15
   4:   81 bf           lr     r11,r15
   6:   4a b9 80 00     xorm   r11,32768
   a:   f0 b0 00 00     c      r11,0 <_ada_hello>
   e:   7b 02           bge    2
  10:   77 08           bex    8
  12:   85 00 00 00     lim    r0,0
  16:   85 10 00 0b     lim    r1,11
  1a:   7e f0 00 00     sjs    r15,0 <_ada_hello>
  1e:   81 fe           lr     r15,r14
  20:   8f ee           popm   r14,r14
  22:   7f f0           urs    r15
```

You can see how big your program is using the **size** command. Note that the sizes are in 8-bit bytes and not 16-bit words.

Example 1.8. Using the Size Program

```
$ m1750-coff-size hello.o
   text    data     bss     dec     hex filename
     62       0       0      62      3e hello.o
$ m1750-coff-size hello
   text    data     bss     dec     hex filename
   9134     744     870   10748    29fc hello
```

To get more detail you can use the object code dump program, and ask for headers. Once again the sizes are in bytes. The addresses are byte addresses.

Example 1.9. Object Code Section Headers

```
$ m1750-coff-objdump -h hello
hello:     file format coff-m1750

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00000036  00000000  00000000  00001000  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  1 .ivec         00000040  00000040  00000040  00001040  2**1
                  CONTENTS, ALLOC, LOAD, READONLY
  2 .text         000021c2  00000100  00000100  00001100  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  3 .rdata        00000176  000022c2  000022c2  000032c2  2**1
                  CONTENTS, ALLOC, LOAD, READONLY
  4 .data         000002e8  00010000  00002438  00004000  2**1
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000366  000102f0  000102f0  00000000  2**1
                  ALLOC
  6 .stab         00004e60  00000000  00000000  000042e8  2**1
                  CONTENTS, DEBUGGING, NEVER_LOAD
  7 .stabstr      00005f3c  00000000  00000000  00009148  2**0
                  CONTENTS, DEBUGGING, NEVER_LOAD
```

1.4. What's in My Program?

You have written five lines of Ada, yet the size command says your program is over 10K bytes. What happened?

Answer: Although we aim to minimize the size of the executable image of your program, there are object code modules that are needed to support the code you've written. Your program has been linked with code from the M1750 Ada libraries. In addition to the application code, the executable program contains the following:

- Program startup code (art0)
- Program elaboration code (adainit)
- Any Ada library packages mentioned in application code with lists (libada)

- Any System packages referenced by the compiler
- Object code from the library libgcc.a, as required
- Object code from other libraries given on the linker command line.

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

```
$ m1750-coff-gnatmake hello.adb -largs -t
m1750-coff-gcc -c hello.adb
m1750-coff-gnatbind -x hello.ali
m1750-coff-gnatlink -t hello.ali
/opt/m1750-ada-1.7/m1750-coff/bin/ld: mode coff_m1750
/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/art0.o
b~hello.o
./hello.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)a-except.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)a-textio.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)a-ioexce.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)x-malloc.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)s-stcosc.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)open.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)close.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)unlink.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)lseek.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)read.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)write.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)sbrk.o
```

In this example, the Text_IO file `a-text_io.o` is 8K bytes in size and accounts for most of the program's 10K bytes.

1.5. Restrictions

Before you go much further, you should be aware of the built-in restrictions. M1750 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 profiles and restrictions, see *The XGC Ada Reference Manual Supplement*.

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 start file and linker script file
- Generating PROM programming files
- Using the debugger
- Using optimizations
- Working on the target
- Checking for stack overflow
- Expanded memory

2.1. How to Customize the Start File

On a real project you will almost certainly need to customize the start file 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 start file `art0.S` contains instructions to initialize the arithmetic unit, floating point unit and system registers. The default start file may be suitable for your requirements. You can see the source code in file `/opt/m1750-ada-1.7/m1750-coff/src/libc/art0.S`. If this is not suitable, make a copy in a working source directory, then edit it as necessary.

Example 2.1. Creating a Custom Start File

```
$ mkdir src
$ cd src
$ cp /opt/m1750-ada-1.7/m1750-coff/src/libc/art0.S myart0.S
$ vi myart0.S
```

When you compile, you should cite the new start file on the command line, as in the following example. We also ask the linker to list all the files that were included in the link.

```
$ $ m1750-coff-gnatmake -f hello -largs -t -nostartfiles myart0.S
m1750-coff-gcc -c hello.adb
m1750-coff-gnatbind -x hello.ali
m1750-coff-gnatlink -t myart0.S -nostartfiles hello.ali
/opt/m1750-ada-1.7/m1750-coff/bin/ld: mode coff_m1750
b~hello.o
./hello.o
/tmp/ccYJNaCH1.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)a-except.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)a-textio.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)a-ioexce.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)x-malloc.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a)s-stcosc.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)open.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)close.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)unlink.o
```

```
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)lseek.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)read.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)write.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a)sbrk.o
```

2.2. Using a Custom 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/m1750-ada-1.7/m1750-coff/lib/ldscripts/coff_m1750.x`. You should copy this file to your local directory, and edit as necessary. See the linker chapter in the XGC User Manual for a description of the format of the file.

Example 2.2. Making a Custom Linker Script File

```
$ cp /opt/m1750-ada-1.7/m1750-coff/lib/ldscripts/coff_m1750.x .
$ mv coff_m1750.x myboard.ld
$ vi myboard.ld
... make any changes ...
```

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

Example 2.3. Using the Custom Linker Script File

```
$ m1750-coff-gnatmake -f hello -largs -T myboard.ld
```

You can add the line “STARTUP(myart0.o)” to your custom linker script file. This will pick up your custom start file object code without having to mention its name of the make command line.

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 **m1750-coff-ld** rather than the Ada linker. Here is an example that generates a map called `hello.map`.

```
$ m1750-coff-gnatmake hello -largs -Wl,-Map=hello.map
```

Example 2.4. The Map File

```
$ more hello.map
...
LOAD /opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/art0.o
LOAD b~hello.o
LOAD ./hello.o
LOAD /opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libada.a
LOAD /opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libgcc.a
LOAD /opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libc.a
LOAD /opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/libgcc.a
                0x00002000                _STACK_SIZE=0x2000
                0x00001000                _ISTACK_SIZE=0x1000
                0x00010000                _PROM_SIZE=0x10000
                0x00010000                _RAM_SIZE=0x10000
                0x00010000                _RAM_START=0x10000
                0x00000000                _PROM_START=0x0
                0x00020000                _RAM_END=(_RAM_START+_RAM_SIZE)
                0x0001fffe                _eistack=(_RAM_END-0x2)
                0x0001f000                _sistack=(_RAM_END-_ISTACK_SIZE)
                0x0001effe                _estack=(_sistack-0x2)
                0x0001d000                _sstack=(_sistack-_STACK_SIZE)
                0x0001cffe                _eheap=(_sstack-0x2)
...lots of output...
```

2.4. Generating PROM Programming Files

By default, the executable file is in Common Object File Format (COFF). Using the object code utility program **m1750-coff-objcopy**, COFF files may be converted into several other industry-standard formats, such as ELF, Intel Hex, and Motorola S Records.

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

```
$ m1750-coff-objcopy --output-target=ihex hello hello.ihex
```

If you don't need the COFF 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 COFF file too.

```
$ m1750-coff-gnatmake -f hello -largs -Wl,-oformat=ihex
$ more hello
:1000000085108000853005268520806AB1219313F4
:100010008530806A910300008513000185208112DC
:10002000B123931385F0F7FFE5EE8500680090009B
...lots of output...
```

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

```
$ m1750-coff-run hello
Hello world
```

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

```
$ m1750-coff-gnatmake -f hello.adb -largs -Wl,-oformat=srec
$ more hello
S008000068656C6C6FE3
S11300008510800085300526852080 6AB1219313F0
S11300108530806A910300008513000185208112D8
S1130020B123931385F0F7FFE5EE850068009000 97
S113003080AE70F001CC00000000000000000000061
S1130040806A8000806E8003807280068076800 9DA
S1130050807A800C807E800F808280218086 80243C
...lots of output...
$ m1750-coff-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 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 **m1750-coff-strip**.

This is how we recompile `hello.adb` with the `-g` option. There are other debug options too. See the *M1750 Ada User's Guide* for more information on debug options.

```
bash$ m1750-coff-gnatmake -f -g hello
m1750-coff-gcc -c -g hello.adb
m1750-coff-gnatbind -x hello.ali
m1750-coff-gnatlink -g hello.ali
```

The debugger is **m1750-coff-gdb**. By default the debugger will run a M1750 program on the M1750 simulator. If you prefer to run and debug on a real M1750 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” [19].

Example 2.5. Running under the Debugger

```
$ m1750-coff-gdb hello
XGC m1750-ada Version 1.7.6 (debugger)
Copyright (c) 1996, 2005, XGC Software.
Based on gdb version 5.1.1
Copyright (c) 1998 Free Software Foundation.
(gdb)break main
Breakpoint 1 at 0x408: file b~hello.adb, line 29.
(gdb)run
Starting program: /home/nettleto/xgc/m1750-ada/examples/hello
Connected to the simulator.
Loading sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         000003fa  00000000  00000000  00001000  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         000005e6  000003fa  000003fa  000013fa  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  2 .rdata        0000006c  000009e0  000009e0  000019e0  2**1
                  CONTENTS, ALLOC, LOAD, READONLY
  3 .data         000000d4  00010000  00000a4c  00002000  2**1
                  CONTENTS, ALLOC, LOAD, DATA
Start address 0x0
Transfer rate: 22784 bits in <1 sec.

Breakpoint 1, main () at b~hello.adb:29
29            adainit;
(gdb)continue
Continuing.
Hello World

Program exited normally.
(gdb)quit
```

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

Example 2.6. Dump of Debug Information

```
bash$ m1750-coff-objdump -G hello

hello:     file format coff-m1750

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String

-1     HdrSym 0      700    00001508 1
0      SO     0      0      00000438 14     /home/opt/m1750-ada-1.7/m1750-coff/src/libada/rts/
1      SO     0      0      00000438 1      x-textio.adb
2      LSYM   0      0      00000000 66     long int:t1=r1;-32768;32767;
3      LSYM   0      0      00000000 95     unsigned char:t2=r2;0;65535;
4      LSYM   0      0      00000000 124    long integer:t3=r1;0020000000000;0017777777777;
...
```

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 norm, and only use other levels of optimization when you want to get something special.

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.

```
$ m1750-coff-gcc -c -O0 whetstone.adb
$ m1750-coff-size whetstone.o
   text    data     bss     dec     hex filename
```

```
  18424       0       0   18424    47f8 whetstone.o
$ m1750-coff-gnatmake -f -O0 whetstone
$ m1750-coff-run whetstone
,.,. Whetstone GTS Version 0.1
---- Floating point benchmark.
Time taken = 2215 mSec
Whetstone rating = 451 KWIPS
```

Here are the results when compiling with optimization level 2. This is the default.

```
$ m1750-coff-gcc -c -O2 whetstone.adb
$ m1750-coff-size whetstone.o
   text    data     bss     dec     hex filename
   9540       0       0    9540    2544 whetstone.o
$ m1750-coff-gnatmake -f -O2 whetstone
$ m1750-coff-run whetstone
,.,. Whetstone GTS Version 0.1
---- Floating point benchmark.
Time taken =     1211 mSec
Whetstone rating =         825 KWIPS
```

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

M1750 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.

Note Note that the monitor is written for a specific target computer and will require customization to work with other target computers.

2.7.1. How to Down-load the Debug Monitor

Before we can use the debugger to down-load and debug programs running on the target, we must down-load the M1750 Ada debug monitor. This is a small program that resides in upper RAM, and communicates with the debugger over a serial interface. You will find the source code in the directory `/opt/m1750-ada-1.7/m1750-coff/src/monitor/`.

```
$ ls /opt/m1750-ada-1.7/m1750-coff/src/monitor/
art1.S      Makefile  remcom.c  xgcmon.c   xgcmon.M
install.sh  README    t1.c      xgcmon.ld
```

In this guide we use the program tip to work as a terminal. This program is generally available on Solaris platforms, but is seldom seen on Linux or Windows. If you don't have tip then there are other programs (such as Kermit) that will do as well.

We configured tip to use the serial interface connected to the target at 19200 bps in the file `dem32`. On Solaris, the configuration statement is in the file `/etc/remote`. The following example shows the configuration line used to generate the rest of this text. Note there is no entry for the output EOF string. This is not required.

The configuration line we use is as follows:

Example 2.7. Remote Configuration File

```
$ cat /etc/remote
...
dem32:\
        :dv=/dev/term/b:br#19200:el=^C^S^Q^U^D:ie=%$:
...
```

The debug monitor is called `xgcmon`. This file is formatted in Motorola S-Records ready for down-loading in response to the load command.

The monitor is now running and ready to communicate over the other serial interface. To leave `tip` type **~.**.

2.7.2. Preparing a Program to Run under the Monitor

Because the debug monitor is a complete supervisor-mode application program it is not appropriate to down-load the programs we built in the previous section. We must rebuild the program using the start file `art1`.

The module `art1` consists of the code from `art0` to do with initializing the high-level language environment. It omits the trap vector and trap handling code. You can get the source from `/opt/m1750-ada-1.7/m1750-coff/src/monitor/art1.S`.

The following code shows how to compile the Ackermann benchmark program using a custom linker script, the module `art1`.

```
$ m1750-coff-gcc -O ackermann.c -o ackermann -T xgcmon.ld art1.o
```

The file `xgcmon.ld` may be found on the CD-ROM in the run-time source directory.

The following example shows the Ackermann benchmark running under the control of the debugger. You should substitute your serial device name for `ttyS0`.

Example 2.8. Remote Debugging

```
$ m1750-coff-gdb ackermann
XGC m1750-ada Version 1.7.6 (debugger)
Copyright (c) 1996, 2005, XGC Software.
Based on gdb version 5.1.1
Copyright (c) 1998 Free Software Foundation.
(gdb) set remote speed 19200
(gdb) tar rem /dev/ttyS0
Remote debugging using /dev/ttyS0
0x21f965c in ?? ()
(gdb) load
Loading section .text, size 0x1948 lma 0x2000000
Loading section .rdata, size 0x3d8 lma 0x2001948
Loading section .data, size 0x50 lma 0x2001d20
Start address 0x2000110
Transfer rate: 6698 bits/sec.
(gdb) run
Starting program: /hdb3/xgc/benchmarks/ackermann
,.,. ackermann GTS Version 0.1
---- ackermann Function call benchmark, A (3, 6).
   - ackermann time taken = 1.130e+00 Seconds.
**** ackermann  PASSED ============================.
Program exited normally.
(gdb) quit
```

2.8. Checking for Stack Overflow

In Version 1.7, stack checks are included by default. To suppress all checks use the compiler option `-gnatp`. The stack limit, which is the lowest address in the stack, is held in global `_stack_limit`. Here is an example that overflows the 8K byte main program stack:

Example 2.9. Stack Overflow Check

```
$ more biggy.adb
procedure Biggy is
   S : String (1 .. 10_000);
begin
   null;
end Biggy;
$ m1750-coff-gnatmake -g biggy
biggy.adb:2:04: warning: "S" is never assigned a value
m1750-coff-gnatbind -x biggy.ali
m1750-coff-gnatlink -g biggy.ali
$ m1750-coff-run biggy
Unhandled exception at AS=0 IC=02E2 (000005c4): Storage_Error
_ada_biggy():
.../examples/biggy.adb:2
```

The global variable _stack_limit is initialized in the run-time system module `art0.S`. The value is computed from the stack bounds declared in the linker script file, and stored with the most significant bit inverted. This is to save instructions when making an unsigned comparison between the limit and the stack pointer.

Note that _stack_limit is set correctly for the main program, for interrupt handlers which use the interrupt stack, and for any Ada tasks, which have their own stacks.

2.9. Expanded Memory

You may distribute the instructions of your program over several address states up to a maximum of 1M word on the 1750A. Calls between compilation units may then need to switch address states. The option for calls that switch address states is **-mlong-calls**, and you must give the option to the compiler, binder and linker. If you are using **gnatmake** then the option **-mlong-calls** must be given after **-cargs** and **-largs**.

```
$ m1750-coff-gnatmake -mlong-calls -f -g hello
m1750-coff-gcc -c -mlong-calls -g hello.adb
```

```
m1750-coff-gnatbind -x hello.ali
m1750-coff-gnatlink -mlong-calls -g hello.ali
```

You can confirm that the expanded memory variants of the library files have been included using the linker's option `-t`, as follows:

```
$ m1750-coff-gnatmake -mlong-calls -f -g hello -largs -Wl,-t
m1750-coff-gcc -c -mlong-calls -g hello.adb
m1750-coff-gnatbind -x hello.ali
m1750-coff-gnatlink -mlong-calls -g -t hello.ali
/opt/m1750-ada-1.7/m1750-coff/bin/ld: mode coff_m1750_expanded
/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/art0.o
b~hello.o
./hello.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libada.a)a-except.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libada.a)a-textio.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libada.a)a-ioexce.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libada.a)x-malloc.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libada.a)s-stcosc.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)open.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)close.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)unlink.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)lseek.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)read.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)write.o
(/opt/m1750-ada-1.7/lib/gcc-lib/m1750-coff/2.8.1/mlong-calls/libc.a)sbrk.o
```

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 the system calls such as `read` and `write`. The C language input-output functions declared in `<stdio.h>` use the same system calls.

With XGC Ada, we have no operating system as such, just the run-time system module art0. However, we support the system call mechanism using the BEX instruction and when running on the simulator we map system calls to host system calls so that

application programs can access host computer files. This is especially useful during program development.

When running on the 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 the library libc and supports an appropriate subset of calls. For example, read and write are directed to UARTA 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/m1750-ada-1.7/m1750-coff/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 2.10. Code to Support Write

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

protected body UART is
   procedure Write (Ch : Character) is
   begin
      --  Code to write one character
   end Write;
end UART;

--  Export pragmas required for compatibility 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
      UART.Write (Ada_Buf (I));
   end loop;

   Result := Count;
end Write;
```

Chapter 3

Real-Time Programs

M1750 Ada is highly suitable 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:

- Ravenscar profile
- The package `Ada.Real_Time` and a high-resolution real-time clock (a precision of one microsecond)
- Preemptive priority scheduling with ceiling locking (120 microsecond task switch[1])
- Low interrupt latency (15 microseconds)
- The packages `Ada.Dynamic_Priorities`, `Ada.Synchronous_Task_Control` and `Ada.Task_Identification`

[1]Simulated generic M1750 at 10 MHz

- Support for periodic tasks and task deadlines, as required by ARINC 653

M1750 Ada also offers reduced program size by:

- Optimized code generation
- Use of trap instructions to raise exceptions
- 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 example real-time programs.

3.1. The Ravenscar Profile

In support of safety-critical applications, Ada 95 offers various restrictions that can be invoked by the programmer to prevent the use of language features that are thought 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 to the Ravenscar profile, you should place the following line at the top of each compilation unit.

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

By default, M1750 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 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

The main subprogram, which contains the entry point, and which is at the root of the compilation unit graph, runs as task number 1. The TCB for this task is created in the run-time system, and the 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 environment 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. Note that the default priority for the main program and for any tasks is 63.

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_set_pwdn` is included in the standard library `libc`. Note that lower power mode requires support from `art0.S` and may not be supportable on your target computer.

Example 3.2. Idle Loop with Power-Down

```
with Built_In_Tests;
procedure T1 is
   pragma Priority (0);
   procedure Power_Down;
   pragma Import (C, Power_Down, "__xgc_set_pwdn");
begin
   loop
      Built_In_Tests.Run;
      Power_Down;
   end loop;
end T1;
```

The rest of the program comprises periodic and aperiodic tasks that are declared in packages, that are with-ed from the main subprogram.

Important In M1750 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. Periodic Tasks

The package `Ada.Real_Time` declares types and subprograms for use by real-time application programs. In M1750 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 10 MHz processor, the lateness of a delay is approximately 55 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 + 55 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 one 10 MHz 1750, you can run a task at 1000Hz, with an overhead (in terms of CPU time) of approximately 20 percent, leaving 80 percent for the application program.

Note A delay statement that gives a time that is already passed has missed its deadline, and will raise a soft deadline fault. The default system call handler logs deadline fault to the console. You may wish to modify this code to log the fault in non-volatile memory.

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 the following example, the task's three scheduling parameters are declared as constants, giving 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 commercial scheduling tool.

Example 3.3. A Periodic Task

```
package body 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;

   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 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 simulator.

The code for the package and its body is given in the following example.

Example 3.4. An Interrupt-Driven Task

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

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

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

   protected IO is
      procedure Handler;
      pragma Attach_Handler (Handler, SPARE2);
      entry Get (C : out Character);
   private
      Rx_Ready : Boolean := False;
   end IO;

   protected body IO is
      procedure Handler is
         Status_Word : Unsigned_16;
      begin
         Asm (Template => "xio    %0,0x8501",
              Outputs => (Unsigned_16'Asm_Output ("=r", Status_Word)),
              Volatile => True);
         Rx_Ready := (Status_Word and 16#0002#) /= 0;
      end Handler;

      entry Get (C : out Character) when Rx_Ready is
         Data_Word : Unsigned_16;
      begin
         Asm (Template => "xio    %0,0x8500",
              Outputs => (Unsigned_16'Asm_Output ("=r", Data_Word)),
              Volatile => True);
         C := Character'Val (Data_Word and 16#007f#);
         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 Example;
```

Points to note are as follows:

- The package Ada.Interrupts.Names declares the names of the M1750 interrupts.
- We use machine code statements to perform IO.
- The type Unsigned_16 permits bitwise operators such as 'and' and 'or'.
- The interrupt handler runs in supervisor mode with the mask register set appropriately for the level of interrupt.

3.2. Additional Predefined Packages

Programs that are not required to follow 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 M1750 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`.

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_A_Rx_Tx);
      --  Must be a parameterless procedure

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

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

   protected body UART_Handler is
      ...
   end UART_Handler;

end Example_Pack;
```

Appendix A Expanded Memory

The M1750 has a sixteen-bit word. In its simplest configuration, the M1750 can address up to 2^{16} or 64K words of memory. With a trivial hardware extension, this can be extended to 64K words of instructions plus 64K words of operands. This is a total of 256K bytes. M1750 Ada supports both of these configurations by default.

Where an application requires more address space than this, the 1750A's addressing range may be extended using a *Memory Management Unit* (MMU). The MMU specified by 1750A offers a further four address bits and allows programs to be up to 1M word in size. This kind of memory is known as “expanded memory”.

A.1. Expanded Memory Solutions

M1750 Ada support two solutions for expanded memory:

- A single program solution using long forms the SJS and URS instructions that switch address states

- A multi-program solution where up to 15 Ada programs may time share the 1750 CPU under the control of a application code in address state zero.

In addition, the programmer is always free to write assembly language statements that either access memory directly, or modify the memory management unit's registers.

A.1.1. The Single-Program Solution

For the single program solution, each application program unit is compiled using the compiler option `-mlong-calls`. This changes the instructions used for subprogram call and subprogram return to the long forms of these instructions, and allocates two words on the stack for the link address. The long forms are actually BEX instructions and call the run-time system to switch address states as necessary.

One the 1750A the maximum memory addressable using this solution is 1M word of instructions with 64K words of data.

With this solution all calls are long calls. In practice the additional time required to make the calls is quite small. The Ackermann benchmark program that consists almost entirely of calls increases in time from 1.18 seconds to 2.62 seconds when running with expanded memory. This suggests that a long call takes roughly twice as long as a normal call.

A.1.2. The Multi-Program Solution

Using the multi-program solution allows the full 1M word to be addressed as instructions or operands. Up to 15 programs may be loaded in parallel, each in its own address state and protected from the others. Address state zero is reserved for a small kernel that supports hard interrupts and the system call interface.

No time-sharing kernel is provided as standard. We expect that application code in address state zero will switch among the loaded programs (according to mission phase for example) and call

functions from those programs with a long call that switches address state.

No special options are used when compiling for this solution as each program executes in a 64K + 64K virtual address space. Also each program is linked as if it were running on non-expanded memory.

Appendix B

M1750 Compiler Options

The compiler can generate code for several different members of the MIL-STD-1750 family and to support expanded memory. The default is the MIL-STD-1750A without expanded memory. For detailed information about the differences, see the draft military standard MIL-STD-1750B, which covers both the 1750A and the 1750B, or see your 1750 vendor's literature.

-mlong-calls
Support expanded memory using the long form of SJS and URS instructions.

-mno-long-calls
Do not support expanded memory. This is the default.

-mb1
Permit 1750B optional mathematical instructions.

-mb2
Permit 1750B optional long loads and stores.

-mb3
Permit 1750B optional unsigned arithmetic and load and store byte instructions.

-mno-b1
Reject 1750B optional mathematical instructions.

-mno-b2
Reject 1750B optional long loads and stores.

-mno-b3
Reject 1750B optional unsigned arithmetic and load and store byte instructions.

Appendix C

M1750 Assembler Options and Directives

This section describes features of the assembler that are specific to the target computer.

C.1. MIL-STD-1750 Options

The assembler can assemble code for several different members of the MIL-STD-1750 family. The default is to assemble code for the MIL-STD-1750A. The following options options control which instructions and addressing modes are permitted. For detailed information about the differences, see the draft military standard MIL-STD-1750B, which covers both the 1750A and the 1750B, or see your 1750 vendor's literature.

-A1750a, **-A1750A**
Assemble for the 1750A with no expanded memory. This is the default.

-A1750b, **-A1750B**
Assemble for the 1750B with all 1750B instruction options but no expanded memory.

-Ama31750, **-AMA31750**
Assemble for the GEC-Plessey MA31750 in 1750B mode.

-Along-calls
Convert LSJS to long call. Convert LURS to long return.

-Ano-long-calls
Convert LSJS to SJS. Convert LURS to URS.

-Ab1
Permit 1750B optional mathematical instructions.

-Ab2
Permit 1750B optional long loads and stores.

-Ab3
Permit 1750B optional unsigned arithmetic, load, and store byte instructions.

-Ano-b1
Reject 1750B optional mathematical instructions.

-Ano-b2
Reject 1750B optional long loads and stores.

-Ano-b3
Reject 1750B optional unsigned arithmetic, load, and store byte instructions.

C.2. Floating Point

The floating directives are as follows:

.float

`Single` precision floating point constants (See MIL-STD-1750A section 4.1.7).

.double

`Double` precision floating point constants (See MIL-STD-1750A section 4.1.6).

C.3. M1750 Machine Directives

The following directives are supported in addition to the common ones listed in the assembler documentation.

.skip *`number`*

.skip is identical to the **.space** directive.

.rdata *`subsection`*

.rdata tells the assembler to assemble the following statements onto the end of the read-only data subsection numbered *`subsection`* (which is an absolute expression). If *`subsection`* is omitted, it defaults to zero.

.rodata *`subsection`*

.rodata is identical to the **.rdata** directive.

.sbam *`flonums`*

.sbam expects one or more flonums, separated by commas. It assembles *Single* precision binary angular measurement (See draft MIL-STD-1750B section 4.1.11).

.dbam *`flonums`*

.dbam expects one or more flonums, separated by commas. It assembles *Double* precision binary angular measurement (See *draft MIL-STD-1750B section 4.1.12*).

C.4. Opcodes

In addition to the opcodes specified in the M1750 Standard, the assembler supports several new ones. These are called pseudo opcodes.

C.4.1. Extended Floating Load Register (ELFR)

The 1750 does not have the important load register instruction for extended precision floating point. The reason is we can copy a three-word extended floating point value from one triple register to another using a single load register and a double load register. However, if the source triple and destination triples overlap, then it is important to get the single and double load in the correct order otherwise the source will be overwritten before it is completely copied.

The opcode EFLR is translated by the assembler into either a single load followed by a double load, or a double load followed by a single load, depending on which registers are used, and guarantees correct operation.

Note that the condition codes will not be correctly set by EFLR. To set the condition codes you should do an extended compare with zero. Of course to check whether a number is negative or not, no matter whether it is a 16 bit, 32 bit or 48 bit, fixed or floating, you only have to test the sign bit of the first word.

C.4.2. Expanded Memory Support

There are two macro-like instructions, LSJS and LURS, for supporting subprogram call and return across address states. Normally these will be translated by the assembler into SJS and URS instructions, but if the assembler is run with the expanded memory option **-Along-calls** then LSJS is expended into a sequence of instructions that makes a call to a subprogram that may be in a different address state and uses the BEX 0 instruction. The LURS instruction is converted into a BEX 2 instruction.

The instruction LLIM is used to load a long (24-bit) byte address into a pair of registers. The address can then be used in the 1750B long load and store instructions, such as LSL and LSS.

C.4.3. Branch Improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that can reach the target address. Generally these mnemonics are made by substituting "`j`" for "`b`" at the start of a standard 1750 mnemonic.

The following table summarizes the pseudo-operations for branches.

Table C.1. M1750 Pseudo Operations for Branches

Pseudo Op	16-bit instruction	32-bit instruction
`j` *label*	`br` *label*	`jc uc,`*label*
`jez` *label*	`bez` *label*	`jc eq,`*label*
`jnz` *label*	`bnz` *label*	`jc ne,`*label*
`jgt` *label*	`bgt` *label*	`jc gt,`*label*
`jlt` *label*	`blt` *label*	`jc lt,`*label*
`jge` *label*	`bge` *label*	`jc ge,`*label*
`jle` *label*	`ble` *label*	`jc le,`*label*

C.4.4. XIO Commands

All the MIL-STD-1750A and 1750B XIO commands are supported. They may be used in both XIO and VIO instructions.

BIT	ITGI	RDOW	RPS
CC	LMP	RFMK	RSW
CI	LOS	RFR	RXMP
CLC	LXMP	RIC1	SFMK
CLIR	MPEN	RIC2	SFR
CO	OD	RIPR	SMK
DMAD	OTA	RLP	SPI
DMAE	OTAR	RMFA	TAH

DSBL	OTB	RMFP	TAS
DSUR	OTBR	RMFS	TBH
ENBL	OTGR	RMK	TBS
ESUR	PI	RMP	TPIO
GO	PO	RNS	WIPR
ICW	RCFR	ROPR	WOPR
ITA	RCS	ROS	WPBS
ITAR	RCW	RPBS	WSW
ITB	RDI	RPI	
ITBR	RDOR	RPIR	

You may write XIO instructions in Ada using the predefined package Machine_Code, as in the following example:

Example C.1. XIO Command in Ada

```
with Machine_Code;
procedure Enable_Interrupts is
   use Machine_Code;
begin
   Asm ("xio    r0, ENBL");
end Enable_Interrupts;
```

C.4.5. Special Characters

There are two special characters used to indicate the start of a comment. These are `'!'` and `'#'`. The line-comment character is `!`. If a `#` appears at the beginning of a line, it is treated as a comment unless it looks like `# line file`, in which case it is treated normally.

Appendix D M1750 Simulator Options

D.1. The Command Line

The simulator command line has the form:

```
$ m1750-coff-run switches files
```

You can get a summary of the options using the help option, as shown in the following example:

Example D.1. Simulator Help

```
$ m1750-coff-run -h
age: m1750-coff-run [options] [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
  -f MOD, --file FILE         Report coverage for this source file only
  -h,   --help                Print this message
  -i I, --pending I           Trigger trace on pending interrupt I
  -I I, --interrupt I         Trigger trace on interrupt level I
  -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
  -p,   --perf                Print performance summary
  -P PC, --pc PC              Trigger trace on pc = PC (use 0x for hex)
  -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
  -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,   --wide                Widen a trace or report
  -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
Simulator options are:
  -freq F           Set clock frequency to F MHz (default 10 MHz)
  -sof              Stop on fault (default)
  -nosof            Don't stop on fault, call handler
  -cpu 1750a        Simulate Generic 1750A (default)
  -cpu ma31750      Simulate Dynex MA31750
  -cpu mas281       Simulate Dynex MAS281
  -cpu pace         Simulate Pace 1750
  -cpu f9450        Simulate Fairchild F9450
  -cpu gvsc         Simulate Honeywell GVSC
IO library options are:
  -uart1 DEV        Connect serial interface 1 to DEV
```

```
  -uart2 DEV        Connect serial interface 2 to DEV
  -slow             Run uarts at 9600 bps (default UNIX speed)
70 col trace format is:
     142.000 cpzn 0 0 1 0000 0000 0000 023456: l      r1,2,r14
            |    | | | |    |    |    |      |  |
            |    | | | |    |    |    |      |  `- Disassembled insn
            |    | | | |    |    |    |      `- Program counter (byte)
            |    | | | |    |    |    `- Pending Interrupt Register
            |    | | | |    |    `- Interrupt Mask
            |    | | | |    `- Fault register
            |    | | | `- Address state (hex)
            |    | | `- Processor state (hex)
            |    | `- Page bank (1750B)
            |    `- Condition codes
            `- CPU time in microseconds
  (status reported before execution)

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

D.2. Command Line Switches

The simulator includes command line switches that are common to all versions, and switches that are specific to the target microprocessor. The more commonly used options are described here.

-v

Verbose Mode. In normal mode the simulator only generates information in the case of an error. In verbose mode, useful information is generated as the simulation proceeds.

-s

When simulation terminates, generates a report that includes execution time, number of clock cycles, and so on.

-t

Trace mode. Use this option to get a continuous listing of instructions as they are processed. The listing includes the execution time so far, the program counter, the instruction processed. If you compiled with the debug option, then the listing will also include source file line numbers.

-d *D*

Delays the start of tracing by *D* microseconds. Use this option to skip unwanted lines of trace output.

-a " *-option -option ...* "

Introduces further target-specific options:

D.3. Examples of Simulator Use

This section contains several example of using the target Microprocessor simulator.

```
$ m1750-coff-run hello
Hello world
```

```
$ m1750-coff-run -s hello
Hello world

-----------------
Statistics Report
-----------------

CPU type:                 Generic 1750A
Clock frequency:          10.0 MHz
Memory allocated:         16384 16-bit words
Instructions executed:    2049
Clock cycles:             7968
Execution time:           796.800 uSec
Average clocks per insn   3.89
M1750 execution speed     2.57 MIPS
```

D.3.1. Tracing Simulation

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

Example D.2. Tracing Simulation

The trace options allow you to get a trace of program execution. In most cases the `-w` option will more information in a wider of longer format. Tracing is triggered either immediately or according to several trigger options. You can trigger on program counter value, after a given number of microseconds, on an interrupt.

```
$ m1750-coff-run -t hello
-----------------------
-- Instruction trace --
-----------------------

Tracing starts at     0.000

microseconds cpznpbpsas   ft   mk   pi     ic  insn
------------+----------+----+----+----+------:-----
       0.000      0 0 0 0000 0000 0000 000000: lim    r1,32768
       0.300    n 0 0 0 0000 0000 0000 000004: lim    r3,4728
       0.600  p   0 0 0 0000 0000 0000 000008: lim    r2,33188
       0.900    n 0 0 0 0000 0000 0000 00000c: sr     r2,r1

...lots of output...

main():
/home/nettleto/xgc/play/b~hello.adb:36
 <main>
     436.700   z  0 0 0 0000 5140 0000 000596: pshm   r14,r14
     437.000   z  0 0 0 0000 5140 0000 000598: lr     r14,r15
     437.200    n 0 0 0 0000 5140 0000 00059a: lr     r11,r15
     437.400    n 0 0 0 0000 5140 0000 00059c: xorm   r11,32768
     437.700  p   0 0 0 0000 5140 0000 0005a0: c      r11,0x000103dc
     438.100  p   0 0 0 0000 5140 0000 0005a4: bge    2
/home/nettleto/xgc/play/b~hello.adb:42
     438.300  p   0 0 0 0000 5140 0000 0005a8: sjs    r15,0x0000056c
adainit():
/home/nettleto/xgc/play/b~hello.adb:8
 <adainit>
     438.700  p   0 0 0 0000 5140 0000 00056c: pshm   r14,r14
     439.000  p   0 0 0 0000 5140 0000 00056e: lr     r14,r15
     439.200    n 0 0 0 0000 5140 0000 000570: lr     r11,r15
     439.400    n 0 0 0 0000 5140 0000 000572: xorm   r11,32768
     439.700  p   0 0 0 0000 5140 0000 000576: c      r11,0x000103dc
     440.100  p   0 0 0 0000 5140 0000 00057a: bge    2
```

```
...lots of output...
```

D.3.2. Tasking Reports

In the following example we can clearly see how the tasking system switched among the 12 tasks that ran during the report's time window. The top line is the idle task and tasks 4 to 12 wait in the ready queue until it's their turn. The second part of the report shows the locking level of the current task in its current protected object.

The following example shows the report for the real-time demonstration program (in directory `demo`) on the M1750 Ada CD-ROM.

Example D.3. Tracing Tasking

```
$ m1750-coff-run -z demo -u 500
...
--------------
Tasking Report
--------------

  'p' PIL set here
  'i' Masked pending interrupt
  'I' Pending interrupt
  '*' Both of the above
  '.' Interrupt unmasked
  '0' The current task with no locks
  '1' The current task with one lock
  ' ' Power down is a blank column

Simulation ended at 1218635.800 uSec

There are no interrupt records
Tasks, y-axis is task number

  't' Current task
  '.' Task in ready queue
  ' ' Task blocked

ttttttttttttttttttttttttttttttttttttttt.......tttttttttttttttttttttttt|  1
                                                                      |  2
                                      ttt                             |  3
                                      ..tt                            |  4
                                      ...tt                           |  5
                                      ....t                           |  6
                                      ....tt                          |  7
                                       ....t                          |  8
                                       ....tt                         |  9
                                       .....t                         | 10
                                       .....tt                        | 11
                                        .....tt                       | 12
+---------+---------+---------+---------+---------+---------+---------0
-35000    -30000    -25000    -20000    -15000    -10000    -5000
                                                  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

                                        111111                        | 127
                                      bbbbbbbbb                       |  10
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb       bbbbbbbbbbbbbbbbbbbbbbbb|   0
+---------+---------+---------+---------+---------+---------+---------0
-35000    -30000    -25000    -20000    -15000    -10000    -5000
                                                  uSec before end time
```

D.3.3. The RAM Tags Report

For each 16-bit word of simulated memory the simulator keeps a tag word that has flags to indicate if the word has been written to, read from, executed and so on. At the end of simulation the simulator can print a report on the tag words. The following example shows the report for the real-time demonstration program (in directory `demo`) on the M1750 Ada CD-ROM.

Example D.4. A RAM Tags Report

```
---------------
RAM Tags Report
---------------

  'S' stack pointer in this block, and block written to
  's' stack pointer in this block
  'X' executed code in the block
  'x' executable code in the block
  'R' read-only data block, has been read
  'r' read-only data block
  'W' block written to
  '.' block unused

This is address state 0

Byte     | Each line represents 4096 bytes
Address  | Each character represents a 64 byte block
--------+-------------------------------------------------------------------
00000000 XR..XXXxXXXXXxxx XXXXXXXXXXXxxXXX XXXXXXxXXXxxXXXX XXXXXXXXXXXXXXXX
00001000 XXXXXXXXXXXXXXXX XXXXXXXXXxXXXxxx xxxxxxXXXXxxxxxx xxXXXXXXXXXXXXXX
00002000 XXXXXXXXxXXXxxxx xxxxxxXXXXXXXXXX XXXxXXXXXXXXxxxx xxxxxxxXXXxxxxxx
00003000 xxxxxxXXXxxxxxxx XXXXxxxxXXXXXXXX XXXXXXXXXXXXXxXx xxxxXXXxxXXXXXXX
00004000 XXXXXxXxxxXXXXXX xXXXxXXXXXXxXXXX XXXRRRRRRRRRRRRR RRRRRRRRRrrrrrrR
00005000 rRrrrrrrRrrrrR.. ................ ................ ................

00010000 WWWWWWWWWWWWWWWW WWWWWWWWWWWWWWWW WWWWWW.......... ................
00011000 ..SSSW.......... ................ ....SSW......... ................
00012000 .....SSWW....... ................ ......SSSW...... ................
00013000 ........SSW..... ................ .........SSW.... ................
00014000 ..........SSW... ................ ...........SSW.. ................
00015000 ............SSSW ................ .............SSS W...............
00016000 ...............S SW.............. ................ SSW.............
00017000 ................ .SSW............ ................ ..SSWW..........
00018000 ................ ...SSSW......... ................ .....SSW........
00019000 ................ ......SSW....... ................ .....SSSS.......

0001e000 ................ ................ ................ ..............SS
0001f000 ................ ................ ................ ..............SS

--------+-------------------------------------------------------------------
```

The main stack, interrupt stack and task stacks are clearly visible and we can see how little they are used.

D.4. How to Customize the Simulator

The simulator's support for the XIO programmed input and output instructions is linked as a sharable library that can be replaced by a compatible user-written library. A template file is provided on the CD-ROM and this can be customized to allow the simulator to interact with other parts of your system, including software simulations of special spacecraft peripheral devices.

You will find the template in `templates/libxio.c`. The CD-ROM file `templates/Makefile` will compile the library using the GNU toolset. If you have some other toolset, or are using GCC with a native linker, consult your manual pages for the appropriate commands and options.

To use the custom library in place of the default one, you must make sure that the directory where you place the custom library is on the library path, and is ahead of the directory that contains the default library. This is easily done by the following statement, which you can place in your login command file.

```
export LD_LIBRARY_PATH=my-directory:$LD_LIBRARY_PATH
```

You can check that the correct library is used in two ways:

- Enter the command **ldd** and check the paths.

```
$ ldd /opt/m1750-ada-1.7/bin/m1750-coff-run
        libncurses.so.5 => /lib/libncurses.so.5 (0x40026000)
        libm.so.6 => /lib/libm.so.6 (0x4006a000)
        libxio.so => libxio.so (0x40088000)
        libc.so.6 => /lib/libc.so.6 (0x4008b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

- Ask the run command for version information. The library will print one line with its version.

```
$ m1750-coff-run -V
XGC m1750-ada Version 1.7 (simulator)
Copyright (c) 1996, 2001, XGC Software.
XGC libxio Version 1.0 (libxio)
Copyright (c) 1996, 2001, XGC Software.
Using BFD version 2.8.3
Copyright (c) 1999 Free Software Foundation.
```

Appendix E

The package Ada.Interrupts.Names

The predefined package `Ada.Interrupts.Names` contains declarations for the M1750 as follows:

```
package Ada.Interrupts.Names is

   --  Hardware interrupts, see MIL-STD-1750A for details

   --     PWRDWN   is handled in art0
   --     MACHERR  is handled in art0
   --     SPARE1   is available to applications
   --     FLVFLOW  maps to Constraint_Error
   --     FXVFLOW  is always ignored
   --     BEX      is handled in art0
   --     FLUFLOW  is always ignored
   --     TIMERA   is shared by applications and art0
   --     SPARE2   is available to applications
   --     TIMERB   is handled in the tasking system
   --     SPARE3   is available to applications
   --     SPARE4   is available to applications
   --     LEVEL1   is available to applications
   --     SPARE5   is available to applications
```

```
   --     LEVEL2   is available to applications
   --     SPARE6   is available to applications

   PWRDWN                 : constant Interrupt_ID := 0;
   MACHERR                : constant Interrupt_ID := 1;
   SPARE1                 : constant Interrupt_ID := 2;
   FLVFLOW                : constant Interrupt_ID := 3;
   FXVFLOW                : constant Interrupt_ID := 4;
   BEX                    : constant Interrupt_ID := 5;
   FLUFLOW                : constant Interrupt_ID := 6;
   TIMERA                 : constant Interrupt_ID := 7;
   SPARE2                 : constant Interrupt_ID := 8;
   TIMERB                 : constant Interrupt_ID := 9;
   SPARE3                 : constant Interrupt_ID := 10;
   SPARE4                 : constant Interrupt_ID := 11;
   LEVEL1                 : constant Interrupt_ID := 12;
   SPARE5                 : constant Interrupt_ID := 13;
   LEVEL2                 : constant Interrupt_ID := 14;
   SPARE6                 : 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;
```

Appendix F

The Host-Target Link

The host-target link allows the debugger to communicate with the debug monitor running on the target computer. 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 connecting cable must include a *null modem*. This is because both the host serial port and target serial port are configured to be connected to a terminal. 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 F.1, “RS-232 Information” [65].

F.1. RS-232 Information

The RS-232 standard is given in Table F.1, “The RS-232 Standard” [66].

Table F.1. 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
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

DB-25	DCE	DB-9			
25				x	Unassigned

The wiring of a null modem cable is given in Table F.2, “Null Modem Wiring and Pin Connection” [67].

Table F.2. 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

Appendix G

Questions and Answers

Here is a list of questions and answers.

Q: How do I change the installation directory? 69
Q: How do I un-install M1750 Ada? 69
Q: Can I do mixed language programming? 70
Q: What is linked into my program over and above my Ada units? 70
Q: Can I build a program with separate code and data areas? 70
Q: Which text editor should I use? 70
Q: Which UNIX shell should I use? 71
Q: Are programs restart-able? 71

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/m1750-ada-1.7/` to that directory.

Q: How do I un-install M1750 Ada?

A: On GNU/Linux, simply delete the directory `/opt/m1750-ada-1.7/` and its contents.

On Solaris, you should use the pkgrm command. For example, M1750 Ada Version 1.7 may be removed as follows:

```
# pkgrm XGCm1ad17
```

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: Which text editor should I use?

A: M1750 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. If you have no favorite editor, then 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
-A1750a and related options, 45
-l option
  M1750, 43, 45
-m1750a and related options, 43

A
architecture options
  M1750, 43, 45

B
branch improvement
  M1750, 49

C
comments
  M1750, 50

D
dbam directive, 47
directives
  M1750, 47
double directive
  M1750, 47

E
EFLR, 48
expanded memory
  M1750, 48

F
fixed point numbers (double), 47
fixed point numbers (single), 47
float directive
  M1750, 46
floating point
  M1750, 46

I
immediate character

M1750, 50
instruction set
M1750, 48

L

line comment character
M1750, 50
LLIM, 48
LSJS, 48
LURS, 48

M

M1750
directives, 47
opcodes, 48
M1750 architecture options, 43, 45
M1750 branch improvement, 49
M1750 expanded memory, 48
M1750 floating point, 46
M1750 immediate character, 50
M1750 line comment character, 50
M1750 opcodes, 48
M1750 options, 43, 45
M1750 pseudo-opcodes, 48, 49
MIL-STD-1750 support, 45

O

opcodes
M1750, 48
options
M1750, 43, 45

P

pseudo-opcodes
M1750, 48, 49

R

rdata directive, 47
rodata directive, 47

S

sbam directive, 47
skip directive, 47
special characters
M1750, 50
string literals, 47

X

XIO commands, 49