Getting Started with M68K Ada

Ada 95 Compilation System for the Motorola M68000 Family

www.xgc.com

Getting Started with M68K Ada

Ada 95 Compilation System for the Motorola M68000 Family

Order Number: M68K-ADA-GS-031214

XGC Technology

London
UK
`<www.xgc.com>`

Getting Started with M68K Ada: Ada 95 Compilation System for the Motorola M68000 Family

Publication date December 14, 2003
© 1998, 1999, 2000, 2001, 2002, 2003, 2004 XGC Software

Acknowledgments

M68K Ada is based on technology originally developed by the GNAT team at New York University and now maintained by Ada Core Technologies, Inc., 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 original mission-critical capability was funded by TRW Aerospace and the Ministry of Defence.

Contents

About this Guide ***xiii***

1 Audience **xiii**
2 Related Documents **xiii**
3 Reader's Comments **xiv**
4 Documentation Conventions **xiv**

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**
  1.3.1 Tracing Simulation **8**
1.4 What's in My Program? **9**
1.5 Restrictions **10**

Chapter 2 *Advanced Techniques* ***13***

2.1 Using a Custom Start File **13**
2.2 Using a Custom Linker Script File **15**
2.3 How to Get a Map File **15**
2.4 Generating PROM Programming Files **16**
2.5 Using the Debugger **18**
2.6 Using Optimizations **20**
2.7 Working with the Target **21**
2.7.1 How to Down-load the Debug Monitor **22**
2.7.2 Preparing a Program to Run under the Monitor **23**

Chapter 3 *Real-Time Programs* ***25***

3.1 The Ravenscar Profile **26**
3.1.1 The Main Subprogram **27**
3.1.2 Periodic Tasks **28**
3.1.3 Form of a Periodic Task **29**
3.1.4 Aperiodic Tasks **30**
3.2 Additional Packages **32**
3.3 Interrupts without Tasks **33**

Appendix A *The M68000 Family* ***35***

Appendix B *Options for the M68000 Family* ***37***

B.1 Compiler Options **37**
B.2 Assembler Options **38**
B.3 Motorola Syntax **41**
B.4 MIT Instruction Syntax **42**
B.5 Floating Point **44**
B.6 Machine Directives **44**
B.7 Opcodes **45**
B.7.1 Branch Improvement **45**
B.8 Linker Options **47**

Appendix C *Using the M68000 Family Simulator* ***49***

C.1 Command Line Switches **49**

Appendix D *The package Ada.Interrupts.Names* ***53***

Appendix E *The Host-Target Link* ***55***

E.1 RS-232 Information **55**

Appendix F *Questions and Answers* ***59***

Index ***63***

Tables

B.1 Assembler Pseudo Operations **45**
E.1 Null Modem Wiring and Pin Connection **56**
E.2 The RS-232 Standard **56**

Examples

1.1 The Source File **2**
1.2 How to Compile `hello.adb` **2**
1.3 Binding and Linking **3**
1.4 Using **gnatmake** **4**
1.5 Running on the Simulator **4**
1.6 Using the gnatmake command **5**
1.7 Generating a Machine Code Listing **6**
1.8 Output from objdump **7**
1.9 Using the Size Command **7**
1.10 Using the Object Code Dump Program **7**
1.11 Tracing Simulation **8**
1.12 A Linker Map **10**
2.1 Creating a Custom Start File **14**
2.2 Recompiling art0.S **14**
2.3 Rebuilding with a Custom art0.S **14**
2.4 Making a Custom Linker Script File **15**
2.5 Using a Custom Linker Script File **15**
2.6 How to Get a Map File **15**
2.7 The Map File **15**
2.8 Converting to Intel Hex **16**
2.9 Generating a HEX File **17**
2.10 Running an Intel Hex File **17**
2.11 Running an S-Record File **17**
2.12 Recompiling with the Debug Option **18**
2.13 Running under the Debugger **19**
2.14 Dump of Debug Information **20**
2.15 Remote Configuration File **22**
2.16 Output from the Monitor **23**
2.17 Remote Debugging **23**
3.1 Main Subprogram with Idle Loop **27**
3.2 Idle Loop with Power-Down **28**
3.3 A Periodic Task **29**
3.4 An Interrupt-Driven Task **30**
3.5 Example Interrupt Level Protected Object **33**

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 *M68K Ada User's Guide* describes the commands, options and scripts required to use the tool-set.

The *M68K 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 compilation systems, are documented in *The XGC Libraries*.

Information on the M68K is available from Motorola, http://www.motorola.com/.

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/].

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 super-user 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 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 your terminal. You will find the source code in the directory `examples` on the M68K Ada CD-ROM.

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

1. The source file(s) must first be *compiled*.
2. The file(s) then must be *bound* using the M68K Ada binder.

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

All three steps are best handled using the **gnatmake** program. Given the name of the main program, **gnatmake** automatically performs the necessary compilation, binding and linking steps. See Section 1.2, “How to Recompile a Program” [4].

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 standard format text file:

Example 1.1. The Source File

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

This file should be named `hello.adb`. Using the default file naming conventions, M68K Ada requires that each file contain 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.

1.1.2. How to Compile

You can compile the program using the following command:

Example 1.2. How to Compile `hello.adb`

```
$ m68k-coff-gcc -c hello.adb
```

The command **m68k-coff-gcc** is used to access the compiler. This command is capable of compiling 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

```
$ m68k-coff-gnatbind hello.ali
$ m68k-coff-gnatlink hello.ali
```

The result is an executable file named `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

```
$ m68k-coff-gnatmake hello
m68k-coff-gcc -c hello.adb
m68k-coff-gnatbind -x hello.ali
m68k-coff-gnatlink hello.ali
$
```

Again the result is an executable file named `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".

Example 1.5. Running on the Simulator

```
$ m68k-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.

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 make utility, 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 **gnatmake** command takes care of these details automatically. Invoke it using one of the following forms:

Example 1.6. Using the gnatmake command

```
$ m68k-coff-gnatmake -v hello
GNATMAKE m68k-ada/-1.7/ Copyright 1995-2001 Free Software Foundation, Inc.
  "hello.ali" being checked ...
  -> "hello.adb" time stamp mismatch
m68k-coff-gcc -c hello.adb
End of compilation
m68k-coff-gnatbind -x hello.ali
m68k-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 `libgnat`. 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 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 M68K Ada Users Guide*, for more information on assembler options.

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

Example 1.7. Generating a Machine Code Listing

```
$ m68k-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 4865 6C6C             .ascii "Hello World"
   6      6F20 576F
   6      726C 64
   7 000b 00                    .even
   8                    .LC1:
   9 000c 0000 0001             .long 1
  10 0010 0000 000B             .long 11
  11                            .text
  12                            .even
  13                    .globl _ada_hello
  14                    _ada_hello:
  15 0000 4E56 0000             link.w  %a6,#0
  16 0004 BBCF                  cmp.l   %sp,%a5
  17 0006 6B02                  bmi.b   .+4
  18 0008 4E45                  trap    #5
  19 000a 203C 0000             move.l  #.LC0,%d0
  19      0024
  20 0010 223C 0000             move.l  #.LC1,%d1
  20      0030
  21 0016 2F01                  move.l  %d1,-(%sp)
  22 0018 2F00                  move.l  %d0,-(%sp)
  23 001a 4EB9 0000             jsr     xgc__text_io__put_line$2
  23      0000
  24 0020 4E5E                  unlk    %a6
  25 0022 4E75                  rts
...
```

You could also use the object code dump utility **m68k-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 1.8. Output from objdump

```
$ m68k-coff-objdump -d hello.o

hello.o:     file format coff-m68k

Disassembly of section .text:

00000000 <_ada_hello>:
   0:   4e56 0000       linkw   %fp,#0
   4:   bbcf            cmpal   %sp,%a5
   6:   6b02            bmis    a <_ada_hello+0xa>
   8:   4e45            trap    #5
   a:   203c 0000 0024  movel   #36,%d0
  10:   223c 0000 0030  movel   #48,%d1
  16:   2f01            movel   %d1,-(%sp)
  18:   2f00            movel   %d0,-(%sp)
  1a:   4eb9 0000 0000  jsr     0 <_ada_hello>
  20:   4e5e            unlk    %fp
  22:   4e75            rts
...
```

You can see how big your program is using the **size** command. The sizes are in bytes.

Example 1.9. Using the Size Command

```
$ m68k-coff-size hello.o
   text    data     bss     dec     hex filename
     56       0       0      56      38 hello.o
$ m68k-coff-size hello
   text    data     bss     dec     hex filename
  10352     420     604   11376    2c70 hello
```

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

Example 1.10. Using the Object Code Dump Program

```
$ m68k-coff-objdump -h hello.o

hello.o:     file format coff-m68k
```

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000024  00000000  00000000  00000104  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data         00000000  00000024  00000024  00000000  2**2
                  ALLOC, LOAD, DATA
  2 .bss          00000000  00000024  00000024  00000000  2**2
                  ALLOC, NEVER_LOAD
  3 .stab         0000015c  00000024  00000024  00000128  2**2
                  CONTENTS, RELOC, DEBUGGING
  4 .stabstr      00000372  00000180  00000180  00000284  2**0
                  CONTENTS, DEBUGGING
  5 .rdata        00000014  000004f2  000004f2  000005f6  2**2
                  CONTENTS, ALLOC, LOAD, READONLY
```

1.3.1. 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 1.11. Tracing Simulation

```
$ m68k-coff-run -t hello
 <__cold_start>
-----------------------
-- Instruction trace --
-----------------------

------------+-------+----+-+-----+--------+---------------------
CPU time in  pending ----psr-----           disassembled
microseconds 7654321 ttsm i xnzvc       pc   instruction
------------+-------+----+-+-----+--------+---------------------
       0.000           s  7       00000100:  braw    0x104
 <start>
       0.160           s  7       00000104:  movew   #112,%sr
       0.320           s  7       00000108:  moveal  #1310716,%sp
art0.S:777
       0.480           s  7       0000010E:  moveq   #0,%d0
       0.560           s  7   z   00000110:  movec   %d0,%cacr
```

```
0.720          s  7   z   00000114:  movec   %d0,%dfc
0.880          s  7   z   00000118:  movec   %d0,%dtt0
1.040          s  7   z   0000011C:  movec   %d0,%dtt1
1.200          s  7   z   00000120:  movec   %d0,%isp
1.360          s  7   z   00000124:  movec   %d0,%itt0
1.520          s  7   z   00000128:  movec   %d0,%itt1
1.680          s  7   z   0000012C:  movec   %d0,%mmusr
1.840          s  7   z   00000130:  movec   %d0,%msp
2.000 Stopped at PC = 0x00000130: interrupted
```

1.4. What's in My Program?

You have written ten lines of Ada, yet the size command says your program is over 5K bytes. What happened?

Answer: Your program has been linked with code from the M68K 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 with-ed 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 link map that lists the object files that have been linked into your program, and the address of every global data item and subprogram.

```
$ m68k-coff-gnatmake hello.adb -largs -Wl,-Map=hello.map
```

Here is part of the link map generated by the last command. We have reduced the width of the map to fit on a page by replacing the path name of the library directory with $p.

Example 1.12. A Linker Map

```
$ more hello.map
Archive member included       because of file (symbol)

$p/libgnat.a(a-except.o) b~hello.o (ada__exceptions___elabs)
$p/libgnat.a(x-textio.o) b~hello.o (xgc__text_io___elabs)
$p/libgnat.a(x-malloc.o) $p/libgnat.a(a-except.o) (__gnat_malloc)
$p/libgnat.a(a-ioexce.o) $p/libgnat.a(x-textio.o) (ada__io_exceptions__use_er
$p/libc.a(schandler.o) $p/art0.o (default_system_call_handler)
$p/libc.a(bcopy.o) $p/libgnat.a(a-except.o) (bcopy)
$p/libc.a(read.o) $p/libgnat.a(x-textio.o) (read)
$p/libc.a(write.o) $p/libgnat.a(x-textio.o) (write)
$p/libc.a(sbrk.o) $p/libgnat.a(x-malloc.o) (sbrk)
$p/libc.a(sys_handler.o) $p/libc.a(schandler.o) (sys_handler)
$p/libc.a(errno.o) $p/libc.a(read.o) (errno)

Allocating common symbols
Common symbol       size    file

__exception_id      0x4     $p/art0.o
__stack_ptr         0x4     $p/art0.o
errno               0x4     $p/libc.a(errno.o)
__exception_pc      0x4     $p/art0.o
xgc__text_io__current_out
                    0x4     $p/libgnat.a(x-textio.o)
...
```

1.5. Restrictions

Before you go much further, you should be aware of the built-in restrictions. M68K 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.

In the main, 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 M68K Ada Technical Summary*.

To set a profile, use the pragma `Profile` as shown in the following example.

```
pragma Profile (Ravenscar);

procedure Main is
...
```

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 with the target

2.1. Using a Custom Start File

The start file `art0.S` contains instructions to initialize the arithmetic unit, floating point unit and timer. The default start file may be suitable for your requirements. You can see the source code in file

`/opt/m68k-ada-1.7/m68k-coff/src/libc/art0.S`. If it is not suitable, make a copy in a working directory, then edit it as necessary.

Example 2.1. Creating a Custom Start File

```
$ mkdir work
$ cd work
$ cp /opt/m68k-ada-1.7/m68k-coff/src/libc/art0.S .
$ vi art0.S
```

One of the configuration parameters you may wish to change is the clock speed. The default speed is 25 MHz, which is the clock frequency of the simulator. If your clock runs at (say) 35 MHz, then you should modify the statement in `art0.S` that defines the clock frequency.

Once you have completed the changes, you must compile `art0.S` to generate an object code file called `art0.0`. This is the file that the default linker script will look for. Note that the compiler will select MC68040 by default. If your target is a MC68000, then use the compile-time option `-m68000`.

The following example gives the command you need:

Example 2.2. Recompiling art0.S

```
$ m68k-coff-gcc -c art0.S
```

If you now rebuild your application program, the local file `art0.0` will be used in preference to the library file. You can check that your local version has been used in the map file.

Example 2.3. Rebuilding with a Custom art0.S

```
$ $  m68k-coff-gnatmake -f -g tt3 -largs -nostartfiles art0.S
```

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 25 MHz.

2.2. Using a Custom Linker Script File

The default linker script file is `/opt/m68k-ada-1.7/m68k-coff/lib/ldscripts/coff_erc.x`. You should copy this file to your local directory, and edit as necessary.

Example 2.4. Making a Custom Linker Script File

```
$ cp /opt/m68k-ada-1.7/m68k-coff/lib/ldscripts/coff_erc.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

```
$ m68k-coff-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 **m68k-coff-ld** rather than the ada linker. Here is an example that generates a map called `hello.map`.

Example 2.6. How to Get a Map File

```
$ m68k-coff-gnatmake hello -largs -Wl,-Map=hello.map
```

Example 2.7. The Map File

```
$ more hello.map
...
 *(.text)
 .text  0x00000104  0x502 art0.o
        0x00000104            __warm_start
        0x00000104            start
 *fill* 0x00000606    0x2
 .text  0x00000608   0x64 b~hello.o
        0x00000632            __break_start
        0x0000065c            ada_main___elabb
        0x00000608            adainit
        0x0000063a            main
        0x0000062a            adafinal
 .text  0x0000066c   0x24 ./hello.o
        0x0000066c            _ada_hello
 .text  0x00000690  0x392 libgnat.a(a-except.o)
        0x00000936            ada__exceptions__save_occurrence
        0x00000690            ada__exceptions___elabs
        0x000009c4            ada__exceptions__save_occurrence$2
        0x00000a10            ada__exceptions___init_proc$2
        0x000008f4            ada__exceptions__exception_information
        0x000006b6            ada__exceptions__exception_message
        0x0000076a            ada__exceptions__reraise_occurrence
        0x000009fe            ada__exceptions___init_proc
        0x000007f8            ada__exceptions__exception_name
        0x000007ca            ada__exceptions__exception_identity
...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 **m68k-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.

Example 2.8. Converting to Intel Hex

```
$ m68k-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.

Example 2.9. Generating a HEX File

```
$ m68k-coff-gnatmake hello -largs -Wl,-oformat=ihex
$ more hello.ihex
:100000000013FFFC00000100000001DC000001DC27
:10001000000001D4000001E4000001E4000001E45C
:10002000000001D400000258000001D400000204C6
:1000300000000260000001DC000001DC0000026042
:100040000000026000000260000002600000026028
:100050000000026000000260000002600000026018
:100060000000020C00000214000002280000023010
:100070000000023800000240000002480000025068
:100080000000026A000002800000028800000045E96
:100090000000047E0000049E000004BE000004DE98
...lots of output...
$
```

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

Example 2.10. Running an Intel Hex File

```
$ m68k-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.

Example 2.11. Running an S-Record File

```
$ m68k-coff-gnatmake -f hello.adb -largs -Wl,-oformat=srec
$ more hello
S008000068656C6C6FE3
S11300000013FFFC00000100000001DC000001DC23
S1130010000001D4000001E4000001E4000001E458
S1130020000001D400000258000001D400000204C2
S113003000000260000001DC000001DC000002603E
S11300400000026000000260000002600000026024
S11300500000026000000260000002600000026014
```

```
S11300600000020C000002140000022800000230OC
S113007000000238000002400000024800000025064
S11300800000026A000002800000028800000045E92
S11300900000047E0000049E000004BE000004DE94
S11300A0000004F2000004FE000004FE000004FE50
S11300B0000004FE000004FE000004FE000004FE34
S11300C0000001E4000001E4000001E4000001E498
...lots of output...
$ m68k-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 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 **m68k-coff-strip**.

This is how we recompile `hello.adb` with the `-g` option. There are other debug options too. See the *M68K Ada User's Guide* for more information on debug options.

Example 2.12. Recompiling with the Debug Option

```
bash$ m68k-coff-gnatmake -f -g hello
m68k-coff-gcc -c -g hello.adb
m68k-coff-gnatbind -x hello.ali
m68k-coff-gnatlink -g hello.ali
```

The debugger is **m68k-coff-gdb**. By default the debugger will run a M68K program on the M68K simulator. If you prefer to run and debug on a real M68K 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” [21].

Example 2.13. Running under the Debugger

```
$ m68k-coff-gdb hello
XGC m68k-ada Version 1.7 (debugger)
Copyright (c) 1996, 2002, XGC Software.
Based on gdb version 5.1.1
Copyright (c) 1998 Free Software Foundation...
(gdb) br main
Breakpoint 1 at 0x644: file b~hello.adb, line 39.
(gdb)run
Starting program: .../examples/hello
Connected to the simulator.

Loading sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .ivec         00000100  00000000  00000000  00002000  2**2
                  CONTENTS, ALLOC, LOAD
  1 .init         00000004  00000100  00000100  00002100  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  2 .text         00001908  00000104  00000104  00002104  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  3 .rdata        000000ec  00001a0c  00001a0c  00003a0c  2**2
                  CONTENTS, ALLOC, LOAD
  4 .data         000001fc  00100000  00001af8  00004000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
Start address 0x104
Transfer rate: 59296 bits in <1 sec.

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

Program exited normally.
[Switching to process 0]
(gdb)quit
```

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

Example 2.14. Dump of Debug Information

```
bash$ m68k-coff-objdump -G hello

hello:     file format coff-m68k

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String

-1     HdrSym 0      1598   00004b7d 1
0      SO     0      0      00000608 13     ../examples/
1      SO     0      0      00000608 1      b~hello.adb
2      LSYM   0      0      00000000 58     long int:t(0,1)=r(0
3      LSYM   0      0      00000000 105    unsigned char:t(0,2
...
```

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.

```
$ m68k-coff-gcc -c -O0 whetstone.adb
$ m68k-coff-size whetstone.o
   text    data     bss     dec     hex filename
  22312       0       0   22312    5728 whetstone.o
```

```
$ m68k-coff-gnatmake -f -O0 whetstone
$ m68k-coff-run whetstone
,.,. Whetstone GTS Version 0.1
---- Floating point benchmark.
Time taken =         325 mSec
Whetstone rating = 3077 KWIPS
```

Here are the results when compiling with optimization level 2.

```
$ m68k-coff-gcc -c -O2 whetstone.adb
$ m68k-coff-size whetstone.o
   text    data     bss     dec     hex filename
  10976       0       0   10976    2ae0 whetstone.o
$ m68k-coff-gnatmake -f -O2 whetstone
$ m68k-coff-run whetstone
,.,. Whetstone GTS Version 0.1
---- Floating point benchmark.
Time taken =         184 mSec
Whetstone rating = 5431 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

M68K 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.

2.7.1. 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 M68K Ada debug monitor. This is a small program that resides in the upper 32K bytes of RAM, and communicates with the M68K debugger over the serial interface B. You will find the source code in the directory `/opt/m68k-ada-1.7/m68k-coff/src/monitor/`.

```
$ ls /opt/m68k-ada-1.7/m68k-coff/src/monitor/
CVS  Makefile  README  art1.S  install.sh  remcom.c  t1.c  t2.adb  t2.c
t6  t6.c  xgcmon.M  xgcmon.c  xgcmon.ld
```

The ready-to-load (S-Record) version is `/opt/m68k-ada-1.7/m68k-coff/lib/xgcmon`.

We assume the target board is fitted with a Motorola monitor.

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 2.15. 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, as shown in the following example.

Example 2.16. Output from the Monitor

```
tbd
```

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/m68k-ada-1.7/m68k-coff/src/monitor/art1.S`.

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

```
$ m68k-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 `/opt/m68k-ada-1.7/m68k-coff/src/monitor`.

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

Example 2.17. Remote Debugging

```
$ m68k-coff-gdb ackermann
XGC m68k-ada Version 1.7b1 (debugger)
Copyright (c) 1996, 2002, XGC Software.
```

```
Based on gdb version 5.1.1
Copyright (c) 1998 Free Software Foundation...
(gdb) set remotebaud 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
$
```

Chapter 3

Real-Time Programs

M68K 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 68040 at 25 MHz

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

M68K 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, M68K 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 Subprogram

The main subprogram, which contains the program entry point runs as task number 1. The TCB[2] for this task is created in the run-time system, and the stack is the main stack declared in the linker script file.

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 using the pragma Priority in the declarative part of the main subprogram.

Example 3.1. Main Subprogram with Idle Loop

```
with My_Packages...
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.

[2]Task Control Block

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`.

Example 3.2. Idle Loop with Power-Down

```
with My_Packages...
procedure T1 is
   pragma Priority (0);
   procedure Power_Down;
   pragma Import (C, Power_Down, "__xgc_set_pwdn");
begin
   --  Confirm successful entry to the main program (say)
   ...

   --  Enter idle loop
   loop
      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. Tasks are numbered from 2 in the order in which they are elaborated.

Important In M68K 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 M68K 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 25 MHz MC68040 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 a 25 MHz M68K, you can run a task at 1000Hz, with an overhead (in terms of CPU time) of approximately 10 percent, leaving 90 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.

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 commercial scheduling tool.

Example 3.3. A Periodic Task

```
T0 : constant Time := Clock;
--  Gets set at elaboration time and used by all periodic tasks

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
      Next_Time := Next_Time + Task1_Period;
      delay until Next_Time;

      --  Do something
      null;
   end loop;
end Task1;
```

The task must have an outer loop the 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 M68K simulator.

Here is the code for the package and its body:

Example 3.4. An Interrupt-Driven Task

```
package EG4_Pack is
   task Task2 is
      pragma Priority (1);
   end Task2;
end EG4_Pack;

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

package body EG4_Pack is
```

```
   use Ada.Interrupts.Names;
   use Interfaces;
   use Text_IO;

   protected IO is
      procedure Handler;
      pragma Attach_Handler (Handler, Level1_Autovector);
      entry Get (C : out Character);
   private
      Rx_Ready : Boolean := False;
   end IO;

   protected body IO is
      procedure Handler is
         Status_Word : Unsigned_8;
         for Status_Word'Address use 16#00050069#;
      begin
         Rx_Ready := (Status_Word and 16#01#) /= 0;
      end Handler;

      entry Get (C : out Character) when Rx_Ready is
         Data_Word : Unsigned_8;
         for Data_Word'Address use 16#00050063#;
      begin
         C := Character'Val (Data_Word and 16#7f#);
         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 EG4_Pack;
```

Points to note are as follows:

- The package Ada.Interrupts.Names declares the names of the M68K external interrupts.
- 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 Interrupt Mask set to the level of the interrupt.

3.2. Additional 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 M68K 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`.

The implementation-defined package `Ada.Periodic_Tasks` allows periodic tasks to be given a period and then run without using the delay statement.

The implementation-defined package `Ada.Task_Deadlines` allows periodic tasks to be given a deadline that can be updated on each iteration. If a task fails to meet a hard deadline, then the program fails with a status that indicates the deadline has been missed.

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 The M68000 Family

The Motorola M68000 Family includes the following members:

MC68000
: First generation 68K processor. 16 bit internal/external data paths. 16 Mb address space.

MC68008
: 8 bit external data path. 1-4 MB address space.

MC68010
: Similar to MC68000, but with restartable instructions. Can be used in a virtual memory environment. Loop mode.

MC68EC000
: Low-power MC68000. 8 or 16 bit external data bus.

MC68020
: 32 bit virtual memory microprocessor. 32 bit internal/external data paths. 4 GB address space. Can be used with floating point coprocessor. New instructions added including bitfield

instructions. New addressing modes added. 256 bytes instruction cache.

MC68EC020
: 16 Mb address space.

MC68030
: Similar to MC68020 but slightly faster. 256 bytes data cache added. On- chip MMU.

MC68EC030
: Low-power MC68030. No MMU.

CPU32
: Basically a 68020 core but without cache, bitfield instructions, and memory indirect addressing modes. 16 bit external data path. No coprocessor. CPU32+ Same as CPU32 but with 32 bit external data path.

MC68040
: Third generation 32 bit processor. 4K instruction cache. 4K data cache. On chip floating point processor. On chip MMU. Most instructions take one cycle.

MC68EC040
: Low-power MC68040. No MMU. No FPU.

MC68060
: Super scalar implementation of the 68K architecture. Can issue up to two instructions per cycle. 8K instruction cache. 8K data cache.

MC68EC060
: Similar to MC68060. No FPU. No MMU.

MC68330, MC68332, MC68340
: Integrated microcontrollers with CPU32.

Appendix B

Options for the M68000 Family

The description of the Ada compiler in *XGC Ada User's Guide* includes information that applies to all target computers. In addition, the compiler offers several target-dependent options that specify which member of the M68000 family is targeted.

The default target is the MC68040. If you wish to target some other member of the 68000 family, then you must do the following:

- Specify the target on the compile command line.
- Create a custom linker script file than specifies the target architecture and machine. The default is `OUTPUT_ARCH(m68k:68040)`.

B.1. Compiler Options

The compiler options that specify a target computer are as follows:

-m68000, **-mc68000**
Generate output for a MC68000, MC68008 or MC68010.

-m68020, **-mc68020**
Generate output for a MC68020.

-m68040, **-mc68040**
Generate output for a MC68040, including the floating point instructions. This is the default.

-mcpu32
Generate code for CPU32 computers, such as the MC68332 and MC68340.

-m68881, **-mc68881**
Generate output containing MC68881 instructions for floating point. Except for the MC68040, in the absence of this option, calls are made to a floating point library (which is not included).

-msoft-float
Generate output containing calls to a floating point library. This is the default with the MC68000 and MC68020.

-mbitfield, **-mno-bitfield**
Generate (don't generate) bit field instructions.

B.2. Assembler Options

The Assembler has several additional command line options as follows:

-l
You can use the **-l** option to shorten the size of references to undefined symbols. If you do not use the **-l** option, references to undefined symbols are wide enough for a full `long` (32 bits). (Since the Assembler cannot know where these symbols end up, the Assembler can only allocate space for the linker to fill in later. Since the Assembler does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only one word wide (16 bits).

This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols are always less than 17 bits away.

--bitwise-or
Normally the character `|` is treated as a comment character, which means that it can not be used in expressions. The **--bitwise-or** option turns `|` into a normal character. In this mode, you must either use C style comments, or start comments with a `#` character at the beginning of a line.

--base-size-default-16, **--base-size-default-32**
If you use an addressing mode with a base register without specifying the size, the Assembler will normally use the full 32 bit value. For example, the addressing mode `%a0@(%d0)` is equivalent to `%a0@(%d0:l)`. You may use the **--base-size-default-16** option to tell the Assembler to default to using the 16 bit value. In this case, `%a0@(%d0)` is equivalent to `%a0@(%d0:w)`. You may use the **--base-size-default-32** option to restore the default behaviour.

--disp-size-default-16, **--disp-size-default-32**
If you use an addressing mode with a displacement, and the value of the displacement is not known, the Assembler will normally assume that the value is 32 bits. For example, if the symbol `disp` has not been defined, the Assembler will assemble the addressing mode `%a0@(disp,%d0)` as though `disp` is a 32 bit value. You may use the **--disp-size-default-16** option to tell the Assembler to instead assume that the displacement is 16 bits. In this case, the Assembler will assemble `%a0@(disp,%d0)` as though `disp` is a 16 bit value. You may use the **--disp-size-default-32** option to restore the default behaviour.

-m68000, **-m68008**, **-m68302**, **-m68306**, **-m68307**, **-m68322**, **-m68356**
Assemble for the MC68000. **-m68008**, **-m68302**, and so on are synonyms for **-m68000**, since the CPUs are the same from the point of view of the assembler.

-m68010
Assemble for the MC68010.

-m68020
Assemble for the MC68020.

-m68030
Assemble for the MC68030.

-m68040
Assemble for the MC68040. This is the default.

-m68060
Assemble for the MC68060.

-mcpu32, **-m68330**, **-m68331**, **-m68332**, **-m68333**, **-m68334**, **-m68336**, **-m68340**, **-m68341**, **-m68349**, **-m68360**
Assemble for the CPU32.

-m5200
Assemble for the ColdFire.

-m68881, **-m68882**
Assemble 68881 floating point instructions. This is the default for the 68020, 68030, and the CPU32. The 68040 and 68060 always support floating point instructions.

-mno-68881
Do not assemble 68881 floating point instructions. This is the default for 68000 and the 68010. The 68040 and 68060 always support floating point instructions, even if this option is used.

-m68851
Assemble 68851 MMU instructions. This is the default for the 68020, 68030, and 68060. The 68040 accepts a somewhat different set of MMU instructions; **-m68851** and **-m68040** should not be used together.

-mno-68851
Do not assemble 68851 MMU instructions. This is the default for the 68000, 68010, and the CPU32. The 68040 accepts a somewhat different set of MMU instructions.

B.3. Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax discussed in Section B.4, “MIT Instruction Syntax” [42]. The Assembler can accept Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction. The two kinds of syntax are fully compatible.

In the following table *`apc`* stands for any of the address registers (`%a0` through `%a7`), the program counter (`%pc`), the zero-address relative to the program counter (`%zpc`), or a suppressed address register (`%za0` through `%za7`). The use of *`size`* means one of `w` or `l`, and it may always be omitted along with the leading dot. The use of *`scale`* means one of `1`, `2`, `4`, or `8`, and it may always be omitted along with the leading asterisk.

The following addressing modes are understood:

Immediate
: *`#number`*

Data Register
: `%d0` through `%d7`

Address Register
: `%a0` through `%a7`@* `%a7` is also known as `%sp`, i.e. the Stack Pointer. `%a6` is also known as `%fp`, the Frame Pointer.

Address Register Indirect
: `(%a0)` through `(%a7)`, `%a7` is also known as `%sp`, the Stack Pointer. `%a6` is also known as `%fp`, the Frame Pointer.

Address Register Postincrement
: `(%a0)+` through `(%a7)+`

Address Register Predecrement
: `-(%a0)` through `-(%a7)`

Indirect Plus Offset
: *`number`*`(%a0)` through *`number`*`(%a7)`, or *`number`*`(%pc)`.

The `number` may also appear within the parentheses, as in `(number,%a0)`. When used with the `pc`, the `number` may be omitted (with an address register, omitting the `number` produces Address Register Indirect mode).

Index
: `number(apc,register.size*scale)`

The `number` may be omitted, or it may appear within the parentheses. The `apc` may be omitted. The `register` and the `apc` may appear in either order. If both `apc` and `register` are address registers, and the `size` and `scale` are omitted, then the first register is taken as the base register, and the second as the index register.

Postindex
: `([number,apc],register.size*scale,onumber)`

The `onumber`, or the `register`, or both, may be omitted. Either the `number` or the `apc` may be omitted, but not both.

Preindex
: `([number,apc,register.size*scale],onumber)`

The `number`, or the `apc`, or the `register`, or any two of them, may be omitted. The `onumber` may be omitted. The `register` and the `apc` may appear in either order. If both `apc` and `register` are address registers, and the `size` and `scale` are omitted, then the first register is taken as the base register, and the second as the index register.

B.4. MIT Instruction Syntax

This syntax for the Motorola M68000 family was developed at the Massachusetts Institute of Technology (MIT).

The Assembler uses instructions names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, `movl` is equivalent to `mov.l`.

In the following table *apc* stands for any of the address registers (`%a0` through `%a7`), the program counter (`%pc`), the zero-address relative to the program counter (`%zpc`), a suppressed address register (`%za0` through `%za7`), or it may be omitted entirely. The use of *size* means one of `w` or `l`, and it may be omitted, along with the leading colon, unless a scale is also specified. The use of *scale* means one of `1`, `2`, `4`, or `8`, and it may always be omitted along with the leading colon.

The following addressing modes are understood:

Immediate
: `#`*number*

Data Register
: `%d0` through `%d7`

Address Register
: `%a0` through `%a7`@* `%a7` is also known as `%sp`, i.e. the Stack Pointer. `%a6` is also known as `%fp`, the Frame Pointer.

Address Register Indirect
: `%a0@` through `%a7@`

Address Register Postincrement
: `%a0@+` through `%a7@+`

Address Register Predecrement
: `%a0@-` through `%a7@-`

Indirect Plus Offset
: *apc*`@(`*number*`)`

Index
: `apc@(`*number*,*register*:*size*:*scale*`)`

  The *number* may be omitted.

Postindex
: *apc*`@(`*number*`)@(`*onumber*`,`*register*`:`*size*`:`*scale*`)`

  The *onumber* or the *register*, but not both, may be omitted.

Preindex
  `apc@(number,register:size:scale)@(onumber)`

  The `number` may be omitted. Omitting the `register` produces the postindex addressing mode.

Absolute
  `symbol`, or `digits`, optionally followed by `:b`, `:w`, or `:l`.

B.5. Floating Point

The floating point formats generated by directives are these.

.float

  `Single` precision floating point constants.

.double

  `Double` precision floating point constants.

.extend, **.ldouble**

  `Extended` precision (`long double`) floating point constants.

Note Packed decimal (P) format floating literals are not supported.

B.6. Machine Directives

In order to be compatible with the Sun assembler the Assembler understands the following directives.

.data1

  This directive is identical to a `.data 1` directive.

.data2

  This directive is identical to a `.data 2` directive.

.even

This directive is a special case of the `.align` directive; it aligns the output to an even byte boundary.

.skip

This directive is identical to a `.space` directive.

B.7. Opcodes

B.7.1. Branch Improvement

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

The following table summarizes the pseudo-operations.

Table B.1. Assembler Pseudo Operations

	Displacement				
Pseudo-Op	**BYTE**	**WORD**	**68020 LONG**	**68000/10 LONG**	**non-PC relative**
jbsr	bsr	bsrw	bsrl	jsr	jsr
jra	bras	bra	bral	jmp	jmp
j*XX*[a]	b*XX*s	b*XX*	b*XX*l	b*NX*s; jmpl	b*NX*s; jmp
db*XX*[a]	db*XX*s	db*XX*	db*XX*; bra; jmpl		
fj*XX*[a]	fb*XX*w	fb*XX*w	fb*XX*l	N/A	fb*NX*w; jmp

[a]XX: condition, NX: negative of condition, see full description below.

jbsr, **jra**

These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target.

j*XX*

Here, `j`*`XX`* stands for an entire family of pseudo-operations, where *`XX`* is a conditional branch or condition-code test. The full list of pseudo-ops in this family is:

jhi	jls	jcc	jcs	jne	jeq	jvc	jvs
jpl	jmi	jge	jlt	jgt	jle		

For the cases of non-PC relative displacements and long displacements on the 68000 or 68010, the Assembler issues a longer code fragment in terms of *`NX`*, the opposite condition to *`XX`*. For example, for the non-PC relative case:

```
jXX foo
```

gives

```
bNXs .L1
        jmp foo
.L1:
```

db*XX*

The full family of pseudo-operations covered here is:

dbhi	dbls	dbcc	dbcs	dbne	dbeq	dbvc	dbvs
dbpl	dbmi	dbge	dblt	dbgt	dble	dbf	dbra
dbt							

Other than for word and byte displacements, when the source reads `db`*`XX`* foo, the Assembler emits

```
dbXX .L1
        bra .L2
.L1:    jmpl foo
.L2:
```

fj*XX*

This family includes the following:

fjne	fjeq	fjge	fjlt	fjgt	fjle	fjf	fjt

fjgl	fjgle	fjnge	fjngl	fjngle	fjngt	fjnle	fjnlt
fjoge	fjogl	fjogt	fjole	fjolt	fjor	fjseq	fjsf
fjsne	fjst	fjueq	fjuge	fjugt	fjule	fjult	fjun

For branch targets that are not PC relative, the Assembler emits:

```
fbNX .L1
        jmp foo
.L1:
```

when it encounters `fjXX foo`.

B.8. Linker Options

The linker script specifies the CPU type of the target computer. The default script specifies the MC68040. Note that the CPU type is written into the executable program so that the simulator knows which CPU to simulate.

OUTPUT_ARCH(m68k:68000)
The target is the MC68000.

OUTPUT_ARCH(m68k:68008)
The target is the MC68008.

OUTPUT_ARCH(m68k:68010)
The target is the MC68010.

OUTPUT_ARCH(m68k:68020)
The target is the MC68020.

OUTPUT_ARCH(m68k:68030)
The target is the MC68030.

OUTPUT_ARCH(m68k:68040)
The target is the MC68040. This is the default.

OUTPUT_ARCH(m68k:cpu32)
The target is any computer that has the CPU32 instruction set.

OUTPUT_ARCH(m68k:68060)
The target is the MC68060.

Appendix C

Using the M68000 Family Simulator

The simulator command line has the form:

```
$ m68k-coff-run switches file
```

C.1. Command Line Switches

The simulator includes command line switches that are common to all targets, and switches that are specific to the target microprocessor.

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

Introduces further target-specific options as follows:

-cpu ***CPU***

The CPU type is given in the program file and the simulator will use this as the CPU type. However, you can change the CPU type to *CPU*. Values are as follows:

- MC68000 - Change to the Motorola MC68000

- MC68008 - Change to the Motorola MC68008
- MC68010 - Change to the Motorola MC68010
- MC68020 - Change to the Motorola MC68020
- MC68040 - Change to the Motorola MC68040
- CPU32 - Change to the Motorola CPU32
- MC68881 - Change to the Motorola MC68881 Floating-Point Co-Processor

-fpu
Simulate the Floating-Point Co-Processor, equivalent to "-cpu MC68881".

-freq *F*
Set the clock frequency to *F* MHz. The default is 25MHz.

-b, **--branch-summary**
Print a branch summary that given the percentage of conditional branches that went both ways. You can use this option to identify blocks of code that have not been covered by a test case.

-B, **--branch-report**
Print a detailed conditional branch report with source line numbers for each conditional branch instruction that did not go both ways. You can use this option to identify blocks of code that have not been covered by a test case.

-c, **--coverage-summary**
Print an execution coverage report that gives the percentage of executable words that were fetched for execution. You can use this option to identify blocks of code that are unreachable from the program entry point.

-C, **--coverage-report**
Print a detailed report giving the source line numbers of executable words that were not fetched for execution. You can

use this option to identify blocks of code that are unreachable from the program entry point.

-d *`D`*, --delay *`D`*
: Delays the start of tracing by *`D`* microseconds. Use this option to skip unwanted lines of trace output.

-h, --help
: Print a list of the simulator's options.

-i *`I`*, --pending *`I`*
: Trigger interrupt trace when interrupt *`I`* is raised (and becomes pending).

-I *`I`*, --interrupt *`I`*
: Trigger interrupt trace when interrupt *`I`* is becomes unmasked and causes the CPU to enter the handler.

-l *`T`*, --limit *`T`*
: Set a time limit on simulation of *`T`* microseconds.

-m, --trace-memory
: Trace memory reads and writes using 70 columns.

-M, --trace-memory-wide
: Trace memory reads and writes using a wide format.

-p, --perf
: Print a performance summary for the simulation run

-r, --ram-tags-report
: Print a report that gives a summary of how each memory block was used. The blocks are large.

-R, --RAM-tags-report
: Print a report that gives a summary of how each memory block was used. The blocks are small.

-s, --stats
: Print execution statistics such as the total number of clock cycles and the number of instructions executed.

-t, **--trace**
Trace instructions using 70 columns.

-T, **--trace-wide**
Trace instructions using wide format and include the floating registers if any have a non-zero value.

-u *U*, **--resolution *U***
Set the task trace resolution to *U* microseconds.

-v, **--verbose**
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.

-V, **--version**
Print the simulator's version number and exit.

-w, **--wide**
Print traces and reports in wide or lengthy format.

-z, **--tasking-report**
Print a tasking report. This records interrupt levels and the number of the current Ada task over a range of time and prints a report when execution is complete. Recording can be triggered on interrupt, or after a delay or it can be continuous. Recording slows the simulator. Uses less than 80 columns.

-Z, **--tasking-report-wide**
Print a tasking report. This records interrupt levels and the number of the current Ada task over a range of time and prints a report when execution is complete. Recording can be triggered on interrupt, or after a delay or it can be continuous. Recording slows the simulator. Uses wide format.

Appendix D

The package Ada.Interrupts.Names

The predefined package `Ada.Interrupts.Names` contains declarations for the M68K as follows:

```
package Ada.Interrupts.Names is

   --  Interrupts from external sources

   Level1_Autovector : constant Interrupt_ID := 1;
   Level2_Autovector : constant Interrupt_ID := 2;
   Level3_Autovector : constant Interrupt_ID := 3;
   Level4_Autovector : constant Interrupt_ID := 4;
   Level5_Autovector : constant Interrupt_ID := 5;
   Level6_Autovector : constant Interrupt_ID := 6;
   Level7_Autovector : constant Interrupt_ID := 7;

   --  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 E 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 serial ports are configured to operate 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 E.1, “RS-232 Information” [55].

E.1. RS-232 Information

The wiring of a null modem cable is given in Table E.1, “Null Modem Wiring and Pin Connection” [56].

Table E.1. 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 is given in Table E.2, “The RS-232 Standard” [56].

Table E.2. 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

DB-25	**DCE**	**DB-9**			
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
25				x	Unassigned

Appendix F

Questions and Answers

Here is a list of questions and answers.

Q: How do I change the installation directory? 59
Q: How do I un-install M68K Ada? 60
Q: Can I do mixed language programming? 60
Q: What is linked into my program over and above my Ada units? 60
Q: Can I build a program with separate code and data areas? 60
Q: Can I use the M68K Boot PROM? 60
Q: Which text editor should I use? 60
Q: Which UNIX shell should I use? 61
Q: Are programs restart-able? 61

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/m68k-ada-1.7/` to that directory.

Q: How do I un-install M68K Ada?

A: On GNU/Linux, simply delete the directory `/opt/m68k-ada-1.7/` and its contents.

On Solaris, you should use the pkgrm command. For example, M68K Ada Version 1.7 may be removed as follows:

```
# pkgrm XGCm6ad17
```

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 M68K Boot PROM?

A: Yes. The program mkprom may be used to create a compressed image and bootstrap loader suitable for the Boot PROM.

Q: Which text editor should I use?

A: M68K 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

A

addressing modes
  M680x0, 43
alternate syntax for the 680x0, 41

B

branch improvement
  M680x0, 45

D

data1 directive
  M680x0, 44
data2 directive
  M680x0, 44
directives
  M680x0, 44
double directive
  M680x0, 44

E

even directive
  M680x0, 45
extend directive M680x0, 44

F

float directive
  M680x0, 44
floating point
  M680x0, 44

I

instruction set
  M680x0, 45

L

ldouble directive M680x0, 44

M

M680x0 addressing modes, 43

M680x0 branch improvement, 45
M680x0 directives, 44
M680x0 floating point, 44
M680x0 opcodes, 45
M680x0 options, 38
M680x0 pseudo-opcodes, 45
M680x0 size modifiers, 42
M680x0 syntax, 42
MIT, 42
Motorola syntax for the 680x0, 41

O

opcodes
  M680x0, 45
options
  M680x0, 38

P

pseudo-opcodes
  M680x0, 45

S

size modifiers
  M680x0, 42
skip directive
  M680x0, 45
syntax
  M680x0, 42