# Rational MC68020/OS-2000 Cross-Development Facility

Document Control Number: 8027A

Rev. 1.0, September 1988
Rev. 1.1, August 1989 (Software Release Rev.4)

This document is subject to change without notice.

Note the Reader's Comments forms at the end of this book, which request the user's evaluation to assist Rational in preparing future documentation.



Rational
3320 Scott Boulevard
Santa Clara, California 95054-3197

# Preface

This *Rational MC68020/OS-2000 Cross-Development Facility* manual presents information for users about a software product that can be added to the basic Rational Environment™. This Cross-Development Facility (CDF) makes it possible to develop and debug application programs for execution in a target environment having an MC68020 CPU running the OS-2000 operating system.

This manual addresses software designers and engineers responsible for implementing target programs using Ada on the Rational® R1000® Development System. The reader is assumed to be knowledgeable about the Ada programming language and the Rational Environment. This manual is intended to be used with the *Rational Environment Reference Manual*.

Chapters 1–3 contain introductory material and brief descriptions of how the CDF is intended to be used. Major individual components of the CDF, which include cross-compiler, cross-assembler, cross-linker, runtime system, downloader, and cross-debugger, are described in Chapters 4–9. Some elements of the CDF (such as the downloader or the command file for the linker) can be additionally enhanced. Appendixes provide detailed reference information that may be of interest to some users, including Appendix F for implementation-specific compiler characteristics, which is required by the *Reference Manual for the Ada Programming Language*.

The information in this manual has been reviewed carefully and is believed to be correct as of the publication date.

If you have questions about using your Rational products, please contact your Rational representative. If you would like to make comments about the usefulness or contents of this manual, please send these to:

Publications Department
Rational
3320 Scott Boulevard
Santa Clara, California 95054-3197

# Contents

# 1 Key Concepts

Often it is necessary or helpful to develop software on one machine (a *host*, such as the Rational R1000) that will execute on another machine (called the *target*). Two classes of programs lend themselves to this approach:

- Programs that will be run on small, embedded targets. Typically, the targets are microprocessors that are components of a larger, more complex system. Such targets are powerful enough to run the application program, but they are not capable of supporting a software-development environment with editors, compilers, and debuggers. Target examples are:

  — MIL-STD-1750A architecture

  — Motorola® 68000-family microprocessors

  These targets require a separate host for the development, debugging, testing, and integration of the software.

- Programs that are designed to run on (one or more) other target hardware architectures and operating systems. The software-development environments of the target computers may not be optimally designed for producing well-engineered code on a tight schedule. Cross-developed software can be designed to run on a variety of targets. Developing software separately for each target could result in different program characteristics for each target. Using one machine as the development environment for multiple targets is an effective way to standardize programs.

Rational's Cross-Development Facility (CDF) products provide users with the ability to develop programs to run on embedded systems or on nonembedded targets. Such programs are developed on the R1000 in its specially designed software-engineering support environment. Instead of generating executable modules designed to run on the R1000, the CDF generates code that can be executed on the designated target machine.

The code produced on the host can be targeted to many different target machines. With care taken to ensure portability, programs developed on a Rational R1000 can be ported to the MIL-STD-1750A architecture, to a bare Motorola 68020 microprocessor environment, to an M68000 target with its own operating system, and/or to a VAX™/VMS™ target. Although the executable modules that actually run on each target machine are different, the source code developed on the host machine can be the same for all of them. Then, if a bug must be fixed, it can be fixed on the universal host and the improved code can be installed on each target.

A typical software-development effort includes the following steps: compile, assemble, link, load, execute, and debug. Depending on the characteristics and facilities supported in the target, it may be desirable to perform some of these steps there. Cross-compilation and debugging are always based on the host, because of its superior capabilities. Therefore, different strategies can be employed for porting code from the universal host to the target machine. For example:

- For embedded architectures, the Ada source code is generated on the universal host, the source code is compiled into object modules on the host, the object modules are linked to become an executable module, and then the executable module is ported to the target machine, where it is executed and debugged from the host. (See Figure 1-1.)

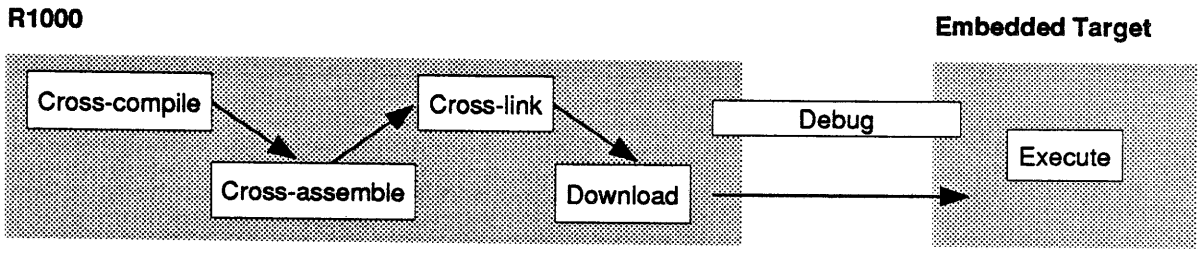



*Figure 1-1 CDF for Embedded Targets*

- For nonembedded targets, some steps in the development process can take place in the target. The advantages of cross-development over target-based software development still apply, but perhaps only the cross-compilation, downloading, and debugging operations need to be conducted from the host environment. (See Figure 1-2.)

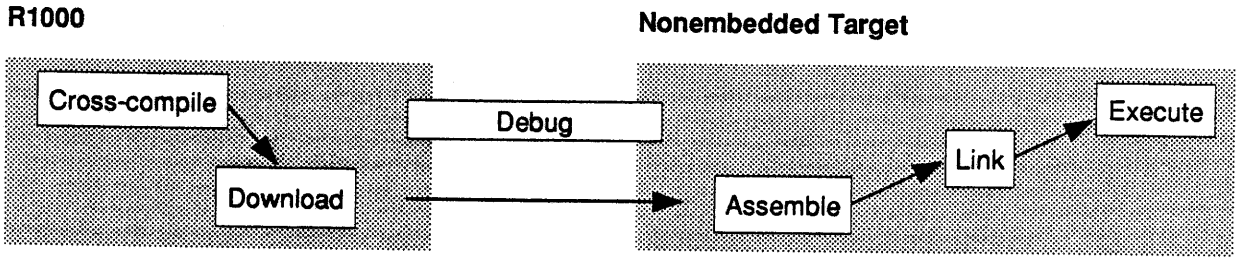



*Figure 1-2 CDF for Nonembedded Targets*

Other distributions of the development stages between host and target are also possible. For any cross-development strategy, an important component is the downloading stage and subsequent debugging of the newly developed programs in their target environment, using debugging facilities from the host.

The Rational CDFs are flexible enough to provide solutions for most cross-development strategies. Each CDF features a cross-compiler that generates target-specific assembly source code.

This source code and any user-originated assembly source can be cross-assembled and cross-linked on the R1000 and then the executable module can be downloaded to the target. For non-embedded targets, the assembly source code can be downloaded to the target and then assembled and linked using the target's native assembler and linker.

The following chapters describe the Rational solution to cross-development of programs to be executed on targets with Motorola 68020 CPUs running the OS-2000 operating system.

# 2 Overview of the MC68020/OS-2000 CDF

The Rational MC68020/OS-2000 Cross-Development Facility (CDF) provides developers working on the R1000 the ability to generate, compile, assemble, link, and debug Ada programs to be executed on a target system with a Motorola 68020 CPU that is running the OS-2000 operating system. The R1000 is a universal host, on which you can create application programs in Ada using the specialized software-engineering facilities of the Rational Environment.

Typically, you will use the CDF to cross-compile source code into MC68020/OS-2000 assembler instructions, assemble these to produce an object module, and then link this object module with other modules (for example, modules from the supplied Ada runtime library) to produce an executable module. The executable module then is converted to an appropriate format and transferred to the target. The resulting code can be executed on the target, with cross-debugging supported from the R1000 host.

The CDF includes all the supporting software you need to accomplish these steps:

- Cross-compile/assemble/link programs targeted to MC68020/OS-2000 processors.
- Write and cross-assemble your own assembly-language programs, and link them with other library modules and/or assembled Ada code into executable modules.
- Download the executable module for target execution.
- Debug your program using the MC68020/OS-2000 cross-debugger.

## MAJOR COMPONENTS

The major components of the MC68020/OS-2000 CDF include:

- Cross-compiler
- Cross-assembler
- Cross-linker
- Runtime library
- File-transfer software
- Cross-debugger

Each of these is described in detail in following chapters of this manual. Another component of the Rational Environment is of special importance to CDF users: configuration management and version control (CMVC). For detailed information about CMVC, see the Project Management (PM) book of the *Rational Environment Reference Manual.*

## COMPILATION MODES

In the Rational Environment, a target key associated with a subsystem or a world sets a compilation mode, indicating to the supporting software the target on which your software will execute. There are more similarities than differences in use of the compilation modes. Because cross-developers typically are experienced with the R1000 compilation mode, the following paragraphs emphasize the differences between this "native" mode and the Mc68020_Os2000 compilation mode.

In either mode, an active Ada unit exists in one of three states:

- Source
- Installed
- Coded

(An Ada unit also can exist in the *archived* state. For a description of this state, see "Ada Images" in the Editing Specific Types (EST) book of the *Rational Environment Reference Manual.*)

The *source* unit contains the Ada code that will be compiled. This code can be edited, more code can be added, and code can be deleted. The Environment provides facilities for syntactic and semantic checking of this code.

A unit in the source state can be promoted to a higher state, the *installed* state. Code in an installed unit has been verified to be syntactically and semantically correct. If this unit depends on other Ada units, the Environment ensures that these units also are installed. Incremental editing and recompilation can be performed on units in the installed state.

An installed unit can be promoted to become a *coded* unit. In this state, the compilation mode makes a significant difference.

### R1000 Compilation Mode

In R1000 compilation mode, a coded unit can be executed immediately. Some incremental operations (such as adding a declaration to a package specification) can occur in the coded state. If other units are required to execute this unit (that is, if they are *with*ed by this unit), the Environment manages the unit dependencies, ensuring that all the units are in the coded state. If a required unit does not exist, or if it cannot be promoted to the coded state, the Environment supplies an appropriate error message.

When you execute an R1000 Ada program consisting of many units, you must know which unit is the main unit and name that unit for execution. Optionally, you may include a pragma Main in the unit, but that inclusion does not directly affect execution. Your program executes on the R1000 hardware, and it uses the supportive Environment tools, such as the R1000 native debugger.

The major features of native compilation on the R1000 include the following:

- The Ada unit exists in a source, installed, or coded state. There are no separate source and object files.
- An Ada unit that is in the coded state is executable. You are not concerned with assembling the object file or linking the necessary files into an executable module.
- The Environment manages all dependencies and guarantees that all the required units are available in the proper state.
- A pragma Main is not required to specify which Ada unit is the main unit.
- The coded units execute on R1000 hardware.

## Mc68020_Os2000 Compilation Mode

In the Mc68020_Os2000 compilation mode, units in the coded state are quite different. Cross-compilation (promotion to the coded state) produces assembly-language source code that is automatically cross-assembled. The resulting output is one relocatable object module for each Ada compilation unit. For program execution on the target, these relocatable object modules need to be linked to become an executable module.

For the cross-linker to produce an executable module for a program, the spec or body of the main Ada unit must contain a pragma Main. This causes the linker to treat that unit as the main program. A main program must be a procedure that is a library unit; it cannot be a function that is a library unit or a subprogram that is in a package.

When a main unit is coded, a relocatable object module for that unit is generated, along with an elaboration module for the entire program. Then the cross-linker is invoked automatically, and it links all of the program's object modules, along with necessary modules from the Ada runtime library, to produce the executable module and a link map that describes the program layout.

The executable module produced by the cross-linker must be converted from the R1000 format to the target format. The converted file must then be transferred to the target hardware, where it can be executed directly using target facilities or indirectly using the cross-debugger that runs on the R1000. The cross-debugger has the same user interface as the R1000 native debugger, and it is capable of additional target-specific debugging operations (such as program stepping at the machine-instruction level).

In summary, the major features of cross-compilation with the MC68020/OS-2000 CDF are these:

- The Ada unit exists in a source, installed, or coded state.
- The Environment manages all dependencies and guarantees that all the required units are available in the proper state.
- The coded state of an Ada unit is not executable. In this state, relocatable object code exists, but it still must be linked.
- A pragma Main is required to specify which Ada unit is the main unit. When the main unit is promoted, it invokes the linker to generate an executable module.
- The executable file must be converted to the target's own object-module format and transferred to the target system.
- The executable module can be executed directly on the target or indirectly using the MC68020/OS-2000 cross-debugger.

## Target Keys

Each world and each path within a subsystem has an associated target key that sets the compilation mode. The target key specifies the target-specific compiler invoked when you enter a compilation command (such as Compilation.Make or Promote) and also the target-specific debugger operations available for the resulting code. The compilers have different code-generator components; they produce target-specific code. You implicitly set the target key when a world or a subsystem is created by your choice of a model world or subsystem. (See Chapter 3 for more information.) Target keys of interest to MC68020/OS-2000 CDF users are:

- R1000 (the default)
- Mc68020_Os2000

Either key selects semantic checking, a code generator, and an appropriate debugger.

# WORLDS

Programs can be developed in one or more worlds, which are components of the R1000 library structure that encapsulates a library's name space. Worlds provide physical and logical separation of library contents from enclosing libraries, which may be worlds or directories. For more information about worlds, see the Library Management (LM) book of the *Rational Environment Reference Manual.*

Using worlds for cross-target development is the same as using worlds for the R1000 target development, except that worlds used for cross-target development must be created with a model world that has the correct target key. Furthermore, library switches must be set up for such worlds. (See "Preparing for MC68020/OS-2000 Development" in Chapter 3 for more details.)

## SUBSYSTEMS

As an alternative to development in worlds, programs can be developed in Rational Subsystems™, which are high-level encapsulations for program components. With subsystems, you can control interfaces between program components, minimize recompilation requirements, and enforce configuration management and version control (CMVC). For details about Rational CMVC and subsystems, see the Project Management (PM) book of the *Rational Environment Reference Manual.*

Subsystems have important advantages over worlds for cross-target development. Subsystems can contain multiple development paths, which are workspaces for building variant implementations of a program. Typically, one path is set up for developing and testing the program on the R1000 target. Then an additional path is created in the same subsystem, where the program can be prepared for the target. The path for the target is created using a model world that has the Mc68020_Os2000 target key. Paths support the development of common code as well as code that is specific to each target.

Using subsystems for cross-target development is very similar to using subsystems for R1000 target development. Typically, the path for the Mc68020_Os2000 target contains load views whose exports are expressed as spec views. However, there are some restrictions:

- If generics will be exported or if subprogram calls from other subsystems must be inlined, the Mc68020_Os2000 target path must be created to contain combined views instead of load views. (Use of combined views requires special release considerations; combined views do not have the same advantages as spec/load views with respect to minimized recompilation requirements and flexible recombinant testing.)
- Within Mc68020_Os2000 paths that contain spec/load views, private parts are not *closed.*
- Currently the capability for generating code views is not available in Mc68020_Os2000 paths.

## MODEL WORLDS

In the Rational Environment, you use a model world to initialize newly created worlds and subsystem paths with:

- A set of external links, which provide visibility to Ada units that reside outside the world or subsystem
- Ada units
- Objects such as switch files

You can create your own model worlds, or you can use default models provided with the Environment (in !Model). The supplied models of interest for the MC68020/OS-2000 CDF are:

- R1000: Includes links for R1000-specific facilities.
- R1000_Portable: Includes links for only those facilities specified by the *Reference Manual for the Ada Programming Language* (LRM) to ensure portability.
- Mc68020_Os2000: Includes links for target-specific facilities and additional Rational-provided facilities for the MC68020/OS-2000.
- Mc68020_Os2000_Portable: Includes links for standard LRM-specified facilities for this target.

# 3 Using the Cross-Development Facility

This chapter describes the steps necessary to create, compile, assemble, link, execute, and debug MC68020/OS-2000 programs in the Rational Environment. The following steps are discussed in the order required to execute a program:

- Preparing the Mc68020_Os2000 environment
- Establishing an Mc68020_Os2000 library switch file
- Creating Ada units
- Compiling, assembling, and linking
- Converting to OS-2000 object-module format
- Transferring the executable file to the target
- Executing and debugging on the target

A sample scenario illustrating these steps is described in the next section; each step is discussed separately in a subsequent section.

## USER SCENARIO

The following scenario illustrates one possible use of the MC68020/OS-2000 Cross-Development Facility (CDF) to develop executable modules for the MC68020/OS-2000 target.

At first, Ada units are developed on the R1000 in a subsystem with the R1000 target key. These units can be edited, tested, and debugged using all the facilities provided by the Rational Environment. The tested code then can be moved to a different subsystem or path that has the Mc68020_Os2000 target key. See later sections in this chapter for more details about subsystems and target keys. The MC68020/OS-2000 cross-compiler is applied to Ada units that have the Mc68020_Os2000 target key.

If required, MC68020/OS-2000 assembly source code can be created and cross-assembled in the same subsystem. See Chapter 5 for details about the MC68020/OS-2000 cross-assembler. To accommodate assembly code:

- Ada units in the program must have the appropriate Interface, Import, or Export pragmas.
- The assembler must be invoked explicitly with the M68k.Assemble command.
- The linker command file must be modified to reference the assembly modules for linking when the main program is compiled. See Chapter 6 for more information about the linker and the linker command files.

When the Ada units are promoted to the coded state, the compilation system automatically selects the appropriate code generator, which generates relocatable object modules. For each unit having a pragma Main, the object modules are linked automatically with required runtime modules (and object modules produced by the cross-assembler, if any) to create the executable module.

Once it is linked, the executable module must be converted to the OS-2000 object-module format with the Convert procedure described in Chapter 8. Finally, the executable module is transferred to the target, where it can be tested by executing commands from a target console or by invoking the MC68020/OS-2000 cross-debugger.

Figure 3-3 depicts a possible user scenario.

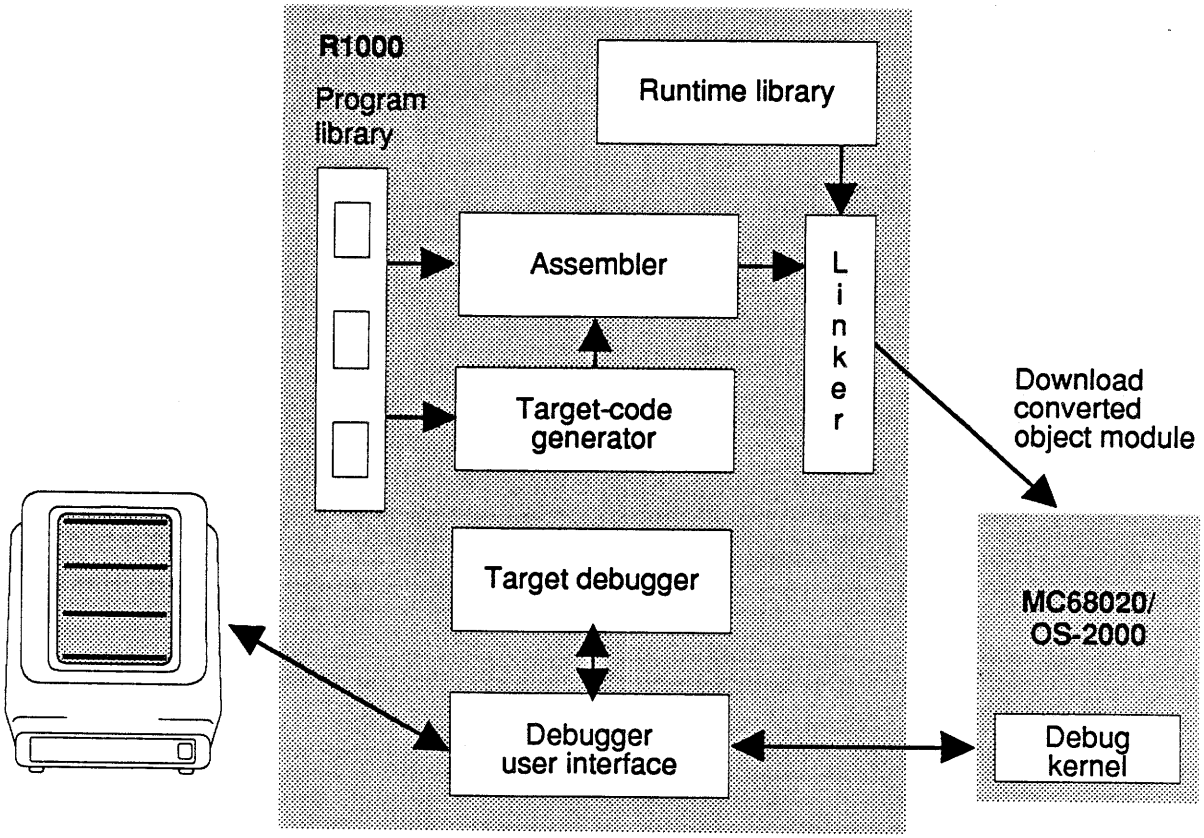



*Figure 3-3 Possible User Scenario*

# PREPARING FOR MC68020/OS-2000 DEVELOPMENT

The first step in preparing a cross-development environment is to choose whether to use subsystems or worlds. It is assumed that you are already familiar with subsystems and worlds; therefore, only the specifics required to operate with the MC68020/OS-2000 CDF will be discussed. For more information on subsystems, see Project Management (PM); for worlds, see Library Management (LM), both in the *Rational Environment Reference Manual.*

Note that programs for an Mc68020_Os2000 target must be developed within subsystem views or worlds that are created for that target. Ada units must be in an Mc68020_Os2000 view or world in order to be cross-compiled for that target.

## Setting Up an Mc68020_Os2000 Path in a Subsystem

Within subsystems, you can compile, assemble, and link your Ada units as well as use the facilities provided by configuration management and version control (CMVC) to manage your project development. Each subsystem can contain multiple paths, one for each target. Typically, you will do most of your development in the working view of an R1000 path and then accept these changes into the working view of an Mc68020_Os2000 path.

To prepare an Mc68020_Os2000 subsystem environment:

1. Use the Cmvc.Initial command to create the desired subsystems, one for each logical program component. Specify models as appropriate so that the initial working view in each subsystem has an R1000 target key. This initial working view defines an R1000 path.
2. Develop and test Ada units in the R1000 path. Use the Cmvc.Make_Controlled command to put these units under CMVC.
3. When desired, create an Mc68020_Os2000 path from the R1000 path. To do this, select the working view of the R1000 path and enter the Cmvc.Make_Path command, specifying at least the following parameters:
   - New_Path_Name: Specify the name prefix for the new path, according to your site's naming conventions. For example, the pathname can indicate the path's target.
   - Model: Specify a model that has an Mc68020_Os2000 target key. Predefined model worlds are described in Chapter 2.
   - Create_Load_View or Create_Combined_View: Specify one of these parameters to indicate whether the new path will contain load or combined views. If Create_Load_View is true, a working load view is created; if Create_Combined_View is true, a working combined view is created. You should use load views if possible; however, you must use combined views if generics are exported or if inlined subprograms are used.
   - Join_Paths: Specify true if all or most of the controlled units are to be shared (joined) between paths; specify false if none or few of the units are to be joined. (When corresponding units are joined across paths, changes to one unit can be propagated automatically to the others in its join set. A joined unit can be checked out in only one path at a time. Unjoined units can be checked out and edited concurrently.)

4. Use the Cmvc.Sever or Cmvc.Join commands as necessary so that all target-independent units are joined between the two paths and target-specific units are severed.

The resulting Mc68020_Os2000 path is a working view that contains a copy of the units from the R1000 path. Development can now continue in either path, as appropriate.

For example, the following command establishes an Mc68020_Os2000_Working path that is joined to Rev1_Working:

```
Cmvc.Make_Path (From_Path => "Rev1_Working",
                New_Path_Name => "Mc68020_Os2000_Working",
                Create_Load_View => True,
                Model => "Mc68020_Os2000",
                Join_Paths => True);
```

This command creates an Mc68020_Os2000 path containing a working load view. All units are joined between the R1000 path and the Mc68020_Os2000 path so that changes made to a unit in one path can be propagated automatically to the corresponding unit in the other path through the Cmvc.Check_Out or Cmvc.Accept_Changes command.

Within the Mc68020_Os2000 path, you can use the Cmvc.Sever command to sever any units containing target-specific code. Severing units allows you to make changes to the units in one view without affecting the corresponding units in the other view.

## Using Mc68020_Os2000 Worlds

Typically, development is done with subsystems. However, if your site does not use subsystems, you must create an Mc68020_Os2000 world. Units cannot be cross-compiled for the target unless they reside in an Mc68020_Os2000 subsystem or world.

### Creating an Mc68020_Os2000 World

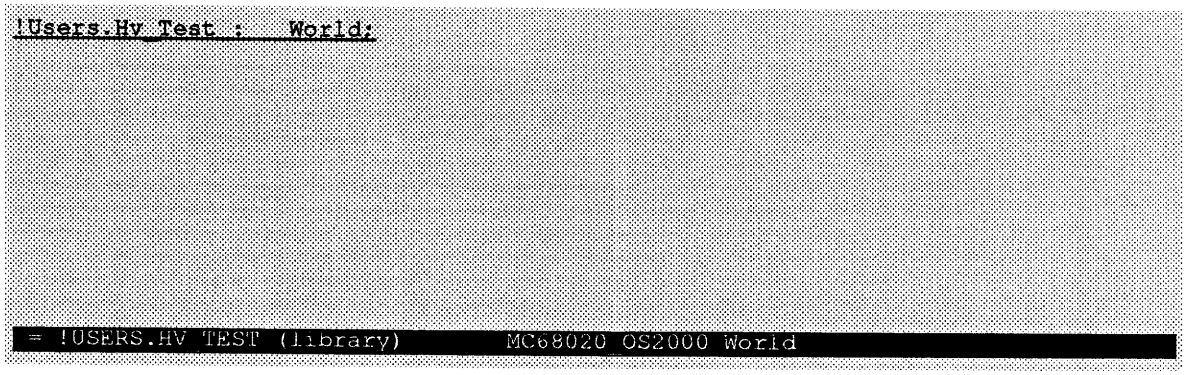
To create an Mc68020_Os2000 world initialized with the proper links:

1. Locate the library that will contain the Mc68020_Os2000 world.
2. Enter the Library.Create_World command, specifying the following parameters:
   - Name: Specify the desired world name.
   - Model: Specify a model that has an Mc68020_Os2000 target key. Predefined model worlds are described in the previous chapter.

For example, the following command creates a world called Hv_Test:

```
Library.Create_World
   (Name => "Hv_Test",
    Model => "!Model.Mc68020_Os2000");
```

The banner for the new world will contain the following legend (note that the target key is specified in the banner):




## CDF LIBRARY SWITCHES

When an Mc68020_Os2000 world is created, a library switch file must be associated with that world if you want to control some of the behavior of the MC68020/OS-2000 compiler, assembler, and linker. (For more information, see package Switches in the Library Management (LM) book of the *Rational Environment Reference Manual.*)

If you are using subsystems, a switch file already exists. You can modify these switches.

### Creating the Switch File

The following steps are required only if you are using worlds; they are not required if you are using subsystems.

A library switch file can be created and associated with the new Mc68020_Os2000 world by one of the following methods:

- *Method 1*

  1. From the new Mc68020_Os2000 world, enter the Switches.Edit command with default parameters.

  A switch file called Library_Switches will be associated with the Mc68020_Os2000 world.

- *Method 2*

  1. From the new Mc68020_Os2000 world, enter the Switches.Create command, specifying the File parameter with the name for the new switch file.

For example, the following command creates a switch file called My_Os2000_Switches:

```
Switches.Create (File => "My_Os2000_Switches");
```

2. Enter the Switches.Associate command, specifying the File parameter with the name of the newly created switch file.

   For example, the following command associates My_Os2000_Switches with a library:

```
Switches.Associate (File => "My_Os2000_Switches",
                    Library => "<IMAGE>");
```

You can now display and/or edit the library switches as necessary.

## Cross-Compiler Switches

The switches of interest for the MC68020/OS-2000 cross-compiler are the Cross_Cg switches. The Cross_Cg switches provided are:

- Asm_Source: Takes a Boolean value; controls retention of the assembly source file generated by the compiler. The filename has an .<Asm> suffix. The default value is false.
- Auto_Assemble: Takes a Boolean value; controls whether the assembly source file that is the result of coding an Ada unit is assembled automatically. Preventing assembly of compilation units prevents generation of executable programs that depend on these compilation units. The default value is true.
- Auto_Link: Takes a Boolean value; controls whether the target linker runs when you code a library procedure body that has an associated pragma Main. The default value is true.
- Debugging_Level: Controls the amount of supporting information that is produced for the debugger when an Ada unit is coded. The possible values are:

  NONE: No debugging information is produced.

  PARTIAL: Debugging tables are produced but optimizations are not inhibited.

  FULL: Debugging tables are produced and the scope of certain optimizations is limited.("Optimizations inhibited" means that code motion across statement boundaries will not occur and the lifetimes of variables will not be reduced.) The default value is FULL.

- Linker_Command_File: Takes a string; a nonnull value of this switch overrides the default filename for the linker command file required to link the object modules into an executable module. The default value is the null string.
- Linker_Cross_Reference: Takes a Boolean value; if true, causes the link map to include cross-reference data that comprises the external symbol name, hex value of its location, module in which it is defined, and modules that reference it. The default value is false.

- Listing: Takes a Boolean value; controls the generation of assembly-language listing files. The filename has the suffix .<List>. The default value is false.
- Optimization_Level: Takes one of the integer values 2, 1, or 0; controls the amount of optimization performed during code generation. The code optimization level represents a trade-off between compiler speed and code quality. This may affect the operation of the debugger and the size of the executable module. The values represent the following:

  2: Fully optimize generated code; this limits effective use of the debugger

  1: Minimally optimize; include only selective inlining for subprogram calls (this is the default

  0: No optimization; slowest compiler operation; but recommended for debugging
- Suppress_All_Checks: Takes a Boolean value; when set to true, it has the same effect as a pragma Suppress_All at the beginning of each Ada unit in the library. The default value is false.

In addition to the Cross_Cg switches, the following Ftp_Profile switches are of interest because they are referenced when the converted executable files are transferred to the target. They also are used by the MC68020/OS-2000 cross-debugger to select a remote machine and a remote directory on the target:

- Remote_Machine: The string value of this switch specifies the FTP name of the default remote machine for file transfers; this should be the target.
- Remote_Directory: The string value of this switch specifies the name of the OS-2000 directory on the target to be used for FTP transfers.

## CREATING ADA UNITS

Now that you have created a subsystem or Mc68020_Os2000 world and have the proper target key and the library switch file associated with the view or world, you are ready to create Ada units. You can create Ada units directly in the new Mc68020_Os2000 view or world, or you can create them in an R1000 view or world and then port them to the Mc68020_Os2000 view or world.

### Creating Ada Units in a Subsystem or an Mc68020_Os2000 World

Creating units in an Mc68020_Os2000 view or world is identical to creating units in an R1000 view or world. You use the Environment facilities to create Ada specs and bodies; [Format] for syntax checking, syntactic completion, and pretty-printing; and [Semanticize] for interactive checking of Ada semantics.

There is one additional requirement in creating Ada units in the Mc68020_Os2000 view or world: you must have a pragma Main at the end of each main unit specification or body. This triggers the invocation of the linker after coding.

## Copying R1000-Developed Ada Units into an Mc68020_Os2000 World

It is suggested that you create, debug, and execute your Ada programs using the features provided by the Rational Environment and, when they are executing properly, move them to an Mc68020_Os2000 view. Care must be taken not to *with* any packages that are specific to the Rational Environment, because these packages are not available on the MC68020/OS-2000 target.

The programs can be copied between worlds using the Library.Copy command. You should set the Copy_Links parameter to false; otherwise, you will copy links to R1000 units.

To copy units from an R1000 to an Mc68020_Os2000 world:

1. From any context, enter the Library.Copy command, specifying the following parameters:
   - From: Specify the pathname of the R1000 world from which units are to be copied.
   - To: Specify the pathname of the Mc68020_Os2000 world into which the units are to be copied.
   - Copy_Links: Change the Copy_Links parameter to false.

For example, the following command transfers the objects to the Mc68020_Os2000 world, where they will be in the source state:

```
Library.Copy
     (From => "!Users.Wjh.R1000_Directory",
      To => "!Project.Start.Mc68020_OS2000_Directory",
      Copy_Links => False);
```

For a more thorough discussion of the Library.Copy procedure, consult the Library Management (LM) book of the *Rational Environment Reference Manual.*

## Porting R1000-Developed Ada Units to an Mc68020_Os2000 Path

If you are using Rational Subsystems, you can create, debug, and functionally test your programs within an R1000 path. Once the programs are executing properly, you can use the CMVC facilities to move the units automatically to an Mc68020_Os2000 path using the Cmvc.Accept_Changes command.

To ensure that the target linker will produce the correct executable module, you must place a pragma Main on the main Ada unit. You can place the pragma Main in the main unit's specification or body before you move the units to the Mc68020_Os2000 view or world, or you can add it into the main unit after you have moved the units.

## COMPILING, ASSEMBLING, AND LINKING ADA PROGRAMS

Now that you have Ada units in your Mc68020_Os2000 view or world, you are ready to compile, assemble, and link them into executable modules. Using the default settings in your library switches (Cross_Cg.Auto_Assemble and Cross_Cg.Auto_Link), you can do all three steps with one key: [Code (This World)].

The following sections describe these three processes and the files associated with them.

### Compiling in an Mc68020_Os2000 View or World

After you have developed your Mc68020_Os2000 Ada units, you can promote them to installed or to coded using the same approach used in R1000 worlds or views—with [Promote], [Code], or [Code (This World)]. Normally, the compilation system invokes the assembler automatically when a compilation unit is promoted to the coded state, and it invokes the linker automatically when a main program is promoted to the coded state.

### Assembling in an Mc68020_Os2000 View or World

Although the compilation system normally invokes the assembler automatically, you can invoke the assembler explicitly if you have special assembly-code files that you want to assemble. If these assembly programs are called by your Ada programs, you must ensure that you use the appropriate pragmas in the Ada code. Each of the following pragmas is placed in the same unit as the Ada specification that contains the Ada declaration corresponding to the assembly-language body:

- Pragma Import_Function
- Pragma Import_Procedure
- Pragma Interface

You now must alter the linker command file so that these assembly modules are included in the executable module. To do this, enter the necessary link commands into the linker command file (refer to Chapter 6, "MC68020/OS-2000 Cross-Linker," for a discussion of the linker command file).

To assemble the units:

1. Enter the M68k.Assemble command, specifying at least the following parameters:
   - Source_File: Specify the name of the input file that contains the assembly source code.
   - Object_File: Specify the name of the output file that will contain the assembly object code.
   - Listing_File: Specify the name of the output file that will contain the assembly listing.

For example, the following command assembles the source in the My_Assembly_Source file into assembler object code, places it in the My_Assembly_Object file, and produces a listing file

called My_Assembly_Listing:

```
M68k.Assemble (Source_File => "My_Assembly_Source",
               Object_File => "My_Assembly_Object
               Listing_File => "My_Assembly_Listing",
               Produce_Listing => True);
```

For a more complete description of this command and its parameters, including default values, see Chapter 5, "MC68020/OS-2000 Cross-Assembler."

## Linking in an Mc68020_Os2000 View or World

The compilation system automatically invokes the linker when you invoke the compiler on a main unit. You can invoke the linker automatically on a user-created linker command file if, for example, you want to include some user-generated assembly files with the output from the compiler. It is strongly suggested that you invoke the linker automatically and use the standard linker command file or use a slightly modified linker command file. For information on the linker command file, consult Chapter 6, "MC68020/OS-2000 Cross-Linker."

If your special requirements necessitate using the linker manually, you can invoke the linker explicitly. In this case, you are responsible for determining the proper linking order and where in memory the program will reside. Also, you must name explicitly the compiler-generated object-code modules and the assembler-generated object-code modules. This will require that you alter the linker command file substantially.

To invoke the linker manually:

1. Enter the M68k.Link command, specifying (for this example) the following parameters:
   - Command_File: Specify the name of the file that contains the linker commands.
   - Exe_File: Specify the name of the file that will contain the executable object module.
   - Produce_Statistics: Specify true if you want statistics.

For example, the following command reads commands in My_Linker_Command_File, produces an executable object module, and sends it to the Main_68k file; it also produces a statistical summary of the number of object modules linked, the number of symbols produced, and the number of fixups required and appends the summary to the link map:

```
M68k.Link (Command_File => My_Linker_Command_File,
           Exe_File => Main_68k,
           Produce_Statistics => True);
```

For a more complete description of this command and all its parameters, including default values, see Chapter 6, "MC68020/OS-2000 Cross-Linker."

## Associated Files

When you compile programs, the compiler produces special files called *associated* files, the names of which appear enclosed with angle brackets (< >) in the library system. If the library switches for these files have been set to true, the files will be retained. These files are associated with their parent (the Ada unit being compiled or assembled). If the parent is deleted or demoted, all the associated files also will be deleted. You cannot create these files directly; they result from invoking the MC68020/OS-2000 CDF. These files have names of the form *file_name.<xxx>*. If you want to have permanent copies of these files that are not associated with their parent, you can copy them into another file, but the new name cannot use the angle brackets (< >).

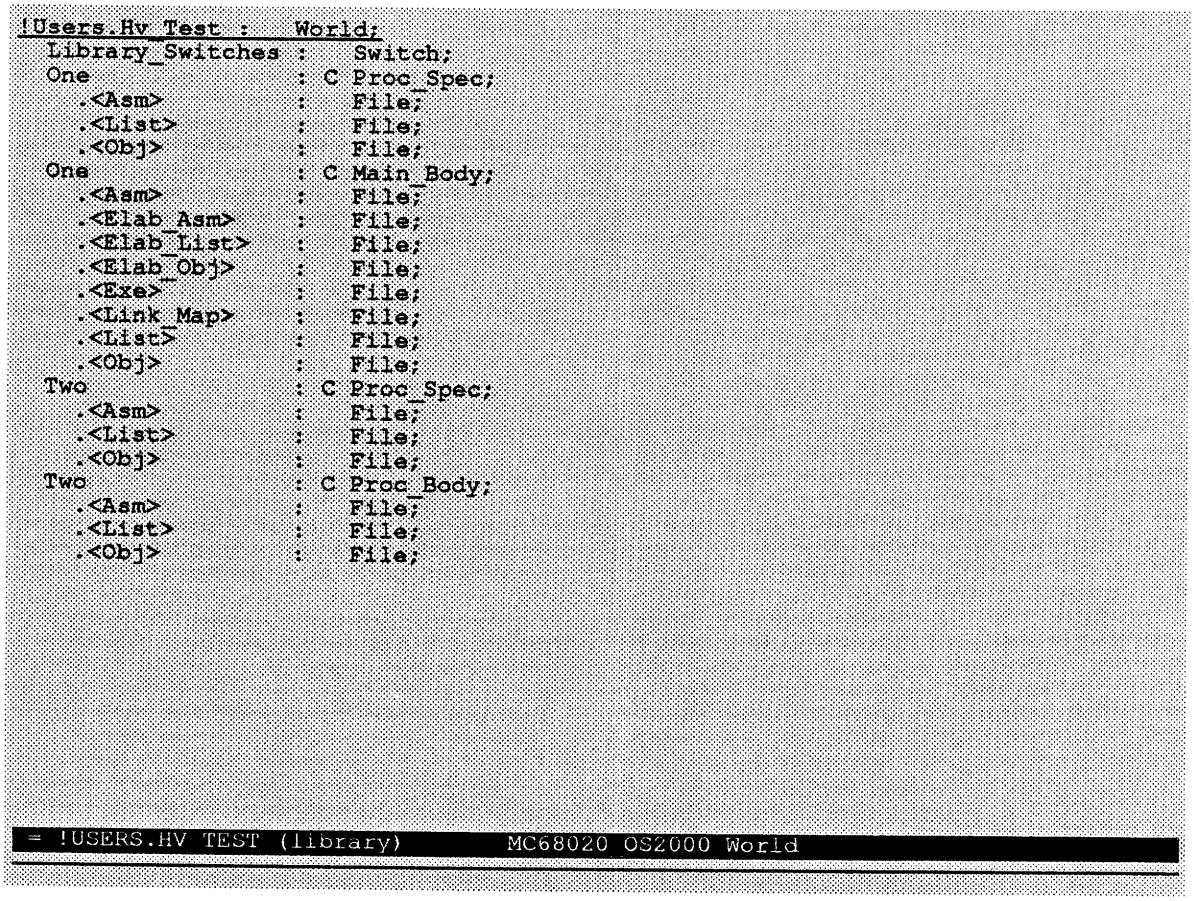
The associated files are:

- <Asm>: Contains the assembly-language source generated by the compiler for the associated compilation unit.
- <List>: Contains the assembly listing generated by the assembler from the <Asm> file.
- <Obj>: Contains the object module generated by the assembler from the <Asm> file. This is a binary file.

The following files are associated only with the main program (the one containing a pragma Main):

- <Elab_Asm>: Contains the assembly-language source generated by the compiler for the main program. The code in this file elaborates each of the units in the Ada program.
- <Elab_List>: Contains the listing for the elaboration code.
- <Elab_Obj>: Contains the object module for the elaboration code.
- <Exe>: Contains the executable module produced by the linker from the <Elab_Obj> file, the <Obj> files of all compilation units in the transitive closure of the associated main program, and the Ada runtime library.
- <Link_Map>: Contains the link map generated by the linker, which describes the <Exe> file associated with the main program.

The following example shows a library that contains the files generated when the MC68020 /OS-2000 CDF is invoked and the Cross_Cg.Listing switch is set to true:




## CONVERTING AND TRANSFERRING EXECUTABLE MODULES

Before you can run executable modules, you must convert them into an object-module format that executes on the target and then transfer them to the target.

### Converting Executable Modules

The executable module produced by the MC68020/OS-2000 linker is in the Rational object-module format. To run on the target, it must be converted to the OS-2000 object-module format.

To convert an executable module:

1. Create a Command window from the library that contains the executable module.
2. Enter **Convert** and press [Complete].

3. Enter the name of the executable module at the **Old_Module** prompt.
4. Enter the name of the executable module to be used on the target at the **New_Module** prompt.
5. Enter **Mc68020_Os2000** at the **New_Format** prompt.
6. Press [Promote].

For example, the following command converts the object-module format:

```
Convert (Old_Module => "Main_68k.<exe>",
         Old_Format => "RATIONAL",
         New_Module => "Main_68k",
         New_Format => "Mc68020_Os2000");
```

For more details about this command and all its parameters, including default values, see Chapter 8, "MC68020/OS-2000 Downloader."

## Transferring Executable Modules

The executable module is now in the OS-2000 object-module format, so you can transfer it to the target.

To transfer an executable module:

1. From the library containing the executable module, create a Command window.
2. Enter **Os2000_Put** and press [Complete].
3. Enter the name of the executable module on the R1000 at the **From_Local_File** prompt.
4. Enter the name of the executable module to be used on the target at the **To_Remote_File** prompt. (If you want to debug this program later, it is recommended that this be the same name as the program on the R1000.)
5. Enter the name of the machine that will receive the executable module at the **Remote_Machine** prompt. (If you have set the Ftp_Profile.Remote_Machine switch in your switch file, you can use the default value for this parameter.)
6. Enter the name of the directory on the machine that will receive the executable module at the **Remote_Directory** prompt. (If you have set the Ftp_Profile.Remote_Directory switch in your switch file, you can use the default value for this parameter.)
7. Press [Promote].

For example, the following command transfers the executable module to the target:

```
Os2000_Put (From_Local_File => "Main_MC68020",
            To_Remote_File => "Main_MC68020",
            Remote_Machine => "<DEFAULT>",
            Remote_Directory => "<DEFAULT>",
            Transliterate => False,
            Profile => Profile.Get);
```

For more information about this command and all its parameters, including default values, see Chapter 8, "MC68020/OS-2000 Downloader."

## EXECUTING AND DEBUGGING

Now that you have produced an executable module and have transferred it to the target, you are ready to run it on the target. This is accomplished by running your program directly on the target with commands from the OS-2000 operating system or by using the R1000-hosted MC68020/OS-2000 cross-debugger.

To execute the program on target hardware:

1. From a console attached to the target, log into the target.
2. Connect to the directory that contains the OS-2000-formatted executable module.
3. Enter the name of the executable module (for example, **`Main_M68k`**) and press [Return].

The program now runs on the target, but it does not run under debugger control.

To enable exception and trap tracing, an additional parameter (-e) can be included with the program name. For example:

```
Main_68k -e
```

To execute the program on target hardware using the MC68020/OS-2000 cross-debugger:

1. From the library containing the selected program, enter the Debug.Invoke command, using default parameters.

   The MC68020/OS-2000 cross-debugger window appears. The debugger is now running and you are ready to execute and debug your program. Source-level debugging is identical to debugging an R1000 program. Program output will appear on the console connected to the target.
2. Press [Execute].

To quit the debugger if the job does not terminate normally, enter the following command:

```
Debug.Kill(Job => True, Debugger => True);
```

This command kills the R1000 and target components of the MC68020/OS-2000 cross-debugger (Debugger => True) and the program being debugged (Job => True). If Debugger => False, the debugging session remains and can be reused, using the Debug.Invoke command.

For more information on source-level commands for debugging, consult the Debugging (DEB) book of the *Rational Environment Reference Manual.* For more information on machine-level debugging on target, see Chapter 9, "MC68020/OS-2000 Cross-Debugger" and the release note.

# 4 MC68020/OS-2000 Cross-Compiler

The MC68020/OS-2000 Cross-Development Facility provides the user with the ability to develop and compile programs using the Rational Environment. Choosing the Mc68020_Os2000 target key selects the MC68020/OS-2000 cross-compiler, which has an MC68020 code generator instead of an R1000 code generator.

The direct output of the Mc68020_Os2000 code generator is MC68020 assembly source code. Under default conditions, this assembly source code is assembled automatically and the resulting object modules are linked automatically into an execution module. Therefore, the differences between the R1000 and Mc68020_Os2000 code generators are not usually visible to the user, and compilation using the MC68020/OS-2000 CDF is identical to compilation with the native compilation system.

## COMPILATION STATES

To compile a program, a user begins with an object, the Ada unit. An active Ada unit exists in one of three states: source, installed, or coded.

Each unit has associated with it a compilation mode, determined by the target key of its world or subsystem. The compilation modes of concern to users of the CDF are the R1000 and Mc68020_Os2000 target keys. Characteristics of a unit depend on both its state and its target key, as indicated in following subsections.

### Source State

In the source state, an Ada unit contains source code that will be compiled. The Ada unit initially is registered with the Environment as an anonymous unit. For example, it might be represented in the world containing it as _Ada_10; this unit cannot be *with*ed by another unit. Once the unit has been promoted to installed, it is registered with the Environment and can be *with*ed by other units. The unit can be demoted back to the source state and still will be registered with the Environment. The source code in either anonymous or registered units can be edited, more source code can be added, and the code can be syntactically and semantically checked. The source state is the same in the R1000 mode and the Mc68020_Os2000 mode.

Units in the source state can be promoted to the next higher state, the installed state. (See "Compiler Commands," in this chapter, for a description of commands that can be used to promote the Ada unit from the source to the installed state.)

## Installed State

The installed state is an intermediate state. In this state, an Ada unit is registered in the world under its package or subprogram name and can now be *with*ed by other Ada units. The Ada unit is syntactically and semantically correct. If this unit depends on other units, they also must be installed. Although users cannot freely add code, delete code, or edit code in units in the installed state, they can perform such operations to a limited extent (see "Incremental Operations," in this chapter). The installed state is the same for the R1000 and the Mc68020_Os2000 targets.

Units in the installed state can be promoted to the next higher state, the coded state, or demoted to the prior state, the source state. (See "Compiler Commands," in this chapter, for a description of commands that can be used to promote an Ada unit from the installed to the coded state or demote it from the installed to the source state.)

## Coded State

In the coded state, the following differences exist between the R1000 compilation mode and the Mc68020_Os2000 mode:

- In the R1000 mode, the Ada unit can now be executed. If the unit *with*s any other Ada units, they also must be in the coded state. If the *with*ed units do not exist or cannot be coded, the Environment provides an appropriate error message. Some incremental operations can be performed in the coded state. (See "Incremental Operations," in this chapter, for a discussion of incremental operations that can be performed in the coded state.)

  A pragma Main optionally may be added to the unit to indicate that it is the main unit.

  Figure 4-1 shows the R1000 compilation mode.

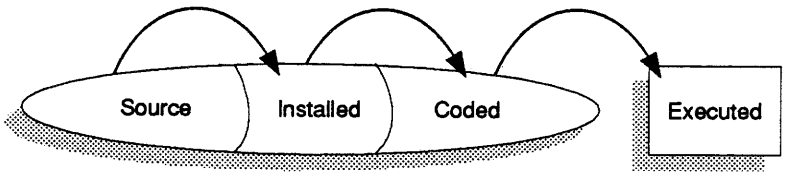



*Figure 4-1 R1000 Compilation Mode*

- In the Mc68020_Os2000 mode, the Ada unit cannot be executed until some additional steps are taken. By default, the output of the MC68020/OS-2000 code generator is assembly source for each Ada library unit. The assembly source is assembled automatically by the assembler into relocatable object modules and the assembly source is deleted. (See Chapter 5,

"MC68020/OS-2000 Cross-Assembler," for more information on the assembly process.) To execute a program, these relocatable object modules must be linked into an executable module by the linker.

When the Ada unit is promoted to the coded state, some associated files are produced. These files may include some or all of the following, depending on the settings of relevant library switches:

| | | |
|---|---|---|
| .<Asm> | .<List> | .<Obj> |
| .<Elab_Asm> | .<Elab_List> | .<Elab_Obj> |
| .<Exe> | .<Link_Map> | |

The library switches also control whether some associated files are retained in the library. See Chapter 3 of this manual or the Library Management (LM) book of the *Rational Environment Reference Manual* for more information about library switches.

Before the executable module can be executed, its object-module format must be changed, and then it must be downloaded to the target (see Chapter 8, "MC68020/OS-2000 Downloader," for more information on the conversion and downloading processes).

Figure 4-2 shows the Mc68020_Os2000 compilation mode.

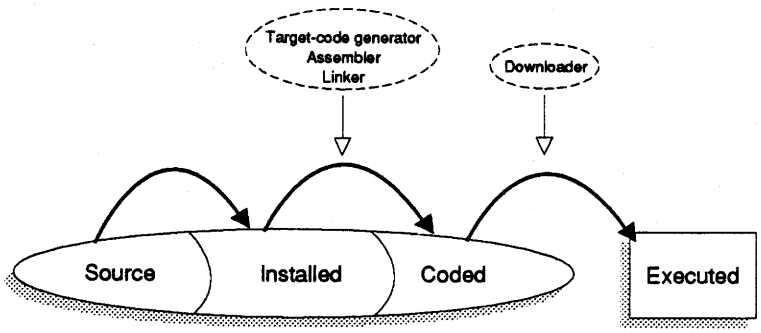



***Figure 4-2 Mc68020_Os2000 Compilation Mode***

## COMPILER COMMANDS

The same commands are used to invoke the MC68020/OS-2000 compiler and the R1000 compiler. The target key associated with the world determines which compiler is invoked. For an extensive discussion of compilation commands, consult the Editing Specific Types (EST) and Library Management (LM) books of the *Rational Environment Reference Manual* or *Rational Environment Basic Operations*.

Table 4-1 summarizes editing and compiler commands typically used with the cross-compiler.

*Table 4-1 Commands Associated with the Compiler*

| Command | Function |
|---|---|
| Common.Abandon | Ends the editing of Ada images. Any changes made to the image since the last commit or promote are lost. However, incremental changes made to installed or coded units, which are permanent as soon as they are promoted, are not lost. |
| Common.Commit | Makes permanent any changes to the Ada image. This procedure is used only for Ada images that are in the source state. When source Ada images are edited, this procedure saves the changes to the image in the underlying permanent representation. The commit operation also is performed implicitly by the Promote, Ada.Code_Unit, and Release procedures. |
| Common.Complete | Completes the selected Ada identifier or the identifiers in the selected element using Ada semantics for name resolution. |
| Common.Create_Command | Creates a Command window below the current Ada window if one does not exist; otherwise, it puts the cursor in the existing Command window below the current window. |
| Common.Definition | Finds the defining occurrence of the designated element and brings up its image in a window on the screen, typically with the definition of the element selected. |
| Common.Demote | Demotes an Ada unit or element to a lower state. |
| Common.Edit | Creates a window in which to edit the named or selected unit and demotes the unit to source if necessary. |
| Common.Enclosing | Finds the parent or enclosing Ada unit of the current window and displays the parent in a window. |
| Common.Explain | Provides an explanation of the error designated by the cursor position in the Ada unit in the current window. Used after semantic or syntactic errors have been discovered, the procedure displays an explanation of those errors in the Message window. |
| Common.Format | Checks the syntax of the current Ada image, performs syntactic completion, and pretty-prints the image. |
| Common.Insert_File | Copies the contents of the text file specified in the Name parameter into the current Ada image at the current cursor position. |

*Table 4-1 Commands Associated with the Compiler (continued)*

| Command | Function |
|---|---|
| Common.Promote | Promotes the Ada image in the current window to the next higher state. |
| Common.Release | Ends the editing of the Ada unit. The unit is unlocked and any changes made to the image are committed (made permanent). |
| Common.Revert | Reverts the Ada image in the current window to the last committed version. |
| Common.Semanticize | Checks the Ada units for semantic correctness. The procedure checks for compliance with the semantic rules of the Ada language. Errors discovered during semantic checking are underlined. |
| Compilation.Atomic_Destroy | Destroys the named objects. |
| Compilation.Compile | Compiles the specified text file into the specified library. This procedure parses and promotes the units in the specified file or files to the specified goal state. |
| Compilation.Delete | Demotes and deletes the default version of the specified object and any subunits. The deletions may be reversible, if the retention count is nonzero. This command differs from the Destroy and Atomic_Destroy procedures, which permanently delete and expunge objects. |
| Compilation.Demote | Demotes the specified unit to the specified goal state, demoting any other units within the limit necessary to achieve the requested demotion. |
| Compilation.Dependents | Displays the set of units that depend on the current or named units. |
| Compilation.Destroy | Destroys the named objects and any subordinate units and demotes dependent units. |
| Compilation.Make | Promotes the specified units to the goal state. By default, this procedure promotes to the coded state the units, their subordinates, and the specs, bodies, and subunits of all units on which they depend. |
| Compilation.Parse | Parses the Ada source in the specified files and creates corresponding Ada units in the specified directory. |
| Compilation.Promote | Promotes the specified unit in the specified scope to the specified goal state. This procedure promotes to the coded state the units, their subordinates, and the specs of all units on which they depend. A unit is not promoted if it is not a legal Ada unit. |

# DIFFERENCES BETWEEN THE COMPILERS

This section briefly highlights the differences between the code generated by the native R1000 compiler for execution on the R1000 and the code generated by the MC68020/OS-2000 cross-compiler for execution on an MC68020 target.

## Chapter 13 Support

Implementation-dependent features of the compilers are different, of course. For more details, see Appendix F of Rational's *Reference Manual for the Ada Programming Language* for the R1000 compiler and Appendix V of this document for the MC68020/OS-2000 cross-compiler.

## Command Windows

Command windows attached to Mc68020_Os2000 worlds behave like Command windows attached to R1000 worlds in that compilations within Command windows reference package Standard for the R1000 and generate R1000 code, not MC68020/OS-2000 code. MC68020/OS-2000 main programs cannot be executed from a Command window. Programs must be run by converting the executable module from R1000 object-module format into OS-2000 format, transferring the converted executable module to the OS-2000 system, and executing on the target.

## Generics

The R1000 architecture supports code-shared generics—multiple instantiations of a generic share the same code. The MC68020/OS-2000 cross-compiler uses macro expansion to implement instantiations of generics, so multiple instantiations yield multiple copies of the code.

## Incremental Operations

In an R1000 world, coded package specifications can be changed incrementally. In an Mc68020_Os2000 world, incremental operations on coded objects are limited to the addition and deletion of comments. The user can perform incremental operations on units in the installed state in an Mc68020_Os2000 world as in an R1000 world.

Table 4-2 shows the object state and the incremental operations that can be performed in that state. For more explanation about performing incremental operations, consult the *Rational Environment Basic Operations* manual.

*Table 4-2 Incremental Operations*

| Object State | Incremental Operation | R1000 | MC68020 |
|---|---|---|---|
| Installed | Add a statement, declaration, or comment. | × | × |
| Installed | Change a statement, declaration, or comment. | × | × |
| Installed | Delete a statement, declaration, or comment. | × | × |
| Coded | Add a comment. | × | × |
| Coded | Insert a declaration into a library-unit package specification. | × | |
| Coded | Change a comment. | × | × |
| Coded | Change a declaration in a library-unit package specification. | × | |
| Coded | Delete a comment. | × | × |
| Coded | Delete a declaration from a library-unit package specification. | × | |

## Packed Records and Arrays

For the R1000, all arrays and records are bit-packed by default. For the Mc68020_Os2000 target, minimization of storage must be requested explicitly with the pragma Pack for a record or array type, or with a record representation specification.

## Record Representation

The R1000 and MC68020/OS-2000 compilers lay out record fields differently.

# 5 MC68020/OS-2000 Cross-Assembler

The MC68020/OS-2000 cross-assembler is an important part of Rational's Cross-Development Facilities, but it is not highly visible to most users. A separate assembler is provided for each target computer family, although all assemblers implement the same target-independent directives, conditional facilities, macro facilities, and object format. The assemblers assemble compiler output, user modules, and user programs.

This chapter addresses individuals writing assembly-language programs or modules. The user should be familiar with assembly-language programming style and techniques, target-specific instructions and instruction syntax, and the Rational Cross-Development Facilities.

Several program examples are provided to illustrate features of the assembler. M68000-family mnemonics and instruction syntax have been used for every example. However, because the assembler is mostly target-independent, these examples apply equally to mnemonics and instruction syntax of any supported target. Knowledge of the M68000 is not required to use this material.

## ASSEMBLER COMMAND (M68k.Assemble)

The compilation system normally invokes the assembler when the Cross_Cg.Auto_Assemble library switch is set to true (the default value). However, it is also possible to invoke the assembler directly by executing:

```
M68k.Assemble (Source_File : String := "<IMAGE>";
               Object_File : String := "<DEFAULT>";
               Listing_File : String := "<DEFAULT>";
               Produce_Object : Boolean := True;
               Produce_Listing : Boolean := True;
               Produce_Statistics : Boolean := False;
               Response : String := "<PROFILE>");
```

The parameters for this command are:

- **`Source_File : String := "<IMAGE>";`**

  Specifies the input file that contains the assembly source code. If <IMAGE> is used, the object in the attached window or the selected object is used as the assembly source file.

Naming expressions also can be used. (Refer to the Library Management (LM) book of the *Rational Environment Reference Manual* for information on wildcards that can be used in naming expressions.) If a naming expression is used, an appropriate target-specific expression also should be used with the Object_File and Listing_File parameters.

- `Object_File : String := "<DEFAULT>";`

  Specifies the output file that will contain the assembly object code. The default appends _Object to the value of the Source_File parameter after removing the _Asm suffix, if it exists.

- `Listing_File : String := "<DEFAULT>";`

  Specifies the output file that will contain the assembly listing. The default appends _List to the value of the Source_File parameter after removing the _Asm suffix, if it exists. The assembly listing can be useful for finding relative addresses of code and data and exact locations of erroneous assembly statements.

- `Produce_Object : Boolean := True;`

  Specifies whether an object file is generated. The default is true. If false is selected, the source file will be checked for correctness, but no object file will be generated.

- `Produce_Listing : Boolean := True;`

  Specifies whether an assembly listing file is generated. The default is true.

- `Produce_Statistics : Boolean := False;`

  Specifies whether statistics of the assembly process are generated. The statistics will be included at the end of the listing file. The default is false.

- `Response : String := "<PROFILE>";`

  Specifies how to respond to errors, how to generate logs, and what activities and switches to use during execution of this command. The default is the job response profile.

## Example

Assume that you want to assemble a file called User_Example. You want the object file to be called User_Example_Object_Code and the listing file to be called User_Example_Listing. You are not interested in generating statistics. The following command accomplishes this:

```
M68k.Assemble (Source_File => "User_Example_Asm",
               Object_File => "User_Example_Object_Code",
               Listing_File => "User_Example_Listing",
               Produce_Listing => True);
```

# ASSEMBLY-LANGUAGE SOURCE CODE

The input to the assembler is a text file, which consists of a series of source statements. Source statements are used to control the assembly process, generate machine instructions, allocate storage, and define constants. The assembler processes source statements one at a time in the order in which they appear.

## Source Statements

### Format

A source statement contains four distinct fields; each is optional. A single statement usually occupies a single line in the source file, but several lines can be used to express a statement by using the line-continuation mechanism. The typical form of a source statement is:

```
[label:][operator[operand{,operand}]][;comment]
```

Source statements are separated by the line terminator—the ASCII character linefeed (16#0A#).

### Label Field

The label field is used to associate a symbolic name with the current value of the location counter. The symbolic name is entered into the user symbol table. A label must conform to the rules for a symbol and must be terminated with the label terminator character, the colon (:). If a label is present within a source statement, it is bound to a value. Although the label may have been introduced previously via a directive or forward reference, it must not have been bound to a value. An attempt to bind a symbol to a value more than once results in an assembly error. The label is associated with the remainder of the source statement only textually, so the following are equivalent, even though the rest of the source statement is on a separate line in the second example:

```
label:  lea     (label),a0

label:
        lea     (label),a0
```

### Operator Field

The operator field can contain an instruction, an assembler directive, or a macro call. This field can be terminated by either a space or a tab. The nature of the operator specifies the context for processing the operand field.

### Operand Field

The operand field contains operands that are specific to the operator. For instance, the operands for an instruction are usually effective addresses, whereas the operand for the .TITLE directive is a character string. When operators require more than one operand, the operands must be separated by commas.

### Comment Field

Comments can be present anywhere in the source file. Comments are separated from the other fields by a semicolon (;). All text between the semicolon and the line terminator is a part of the comment field and is ignored. No restrictions are placed on the characters within a comment.

### Continuation Lines

Normally a single source statement is contained on a single source line. In some cases, it may be desirable to use several source lines to express a single source statement. To accomplish this, a line-continuation character is required to indicate that the end of the line is not the end of the source statement. The line-continuation character is the vertical bar (|). All characters between the bar and the end-of-line character are ignored.

```
move.l ([table,d0*4],table_entry_offset),   | this is a comment
       ([table,d1*4],table_entry_offset)
```

## Numeric Literals

Numeric literals consist of two forms: unbased and based. Unbased literals are evaluated in the current radix. Based literals define a radix for evaluation using Ada syntax. The default radix is 10. The syntax for based constants is:

```
radix#numeric_literal#
```

The base is evaluated in the decimal radix and may be 2, 8, 10, or 16:

```
.radix  10       ; change the current radix to decimal
.dc.b   100      ; this is 100 decimal
.dc.b   16#100#  ; this is 256 decimal
.radix  16       ; change the current radix to hex
.dc.b   100      ; this is 256 decimal
.dc.b   10#100#  ; the radix is decimal; this is 100 decimal
.dc.b   16#100#  ; the base is 16 decimal; this is 256 decimal
```

To distinguish numeric literals from identifiers, all numeric literals must begin with a digit:

```
.radix  10#16#   ; make the current radix hex
.dc.l   ff       ; this is a reference to the symbol FF
.dc.l   0ff      ; this is a numeric constant equal to 255 decimal
.dc.l   16#ff#   ; this is never ambiguous
```

## Symbols

Symbols are used to equate a name with a value. Symbols are strings of 1 through 32 characters, as specified below. Symbols that exceed the 32-character limit are flagged as illegal.

### Symbol Character Set

The following characters may appear within the text of a symbol:

| | |
|---|---|
| A–Z | Letters of the alphabet (case-insensitive) |
| 0–9 | Decimal digits |
| _ | Underscore |
| . | Period |
| $ | Dollar sign |
| ' | Apostrophe |

Other characters are illegal within a symbol.

### Symbol Types

Symbol types include:

- Permanent symbols
- User-defined permanent symbols
- User-defined temporary symbols
- Macro name symbols

*Permanent symbols* include instruction mnemonics, register symbols, assembler directives, and one special symbol, the period (.), which is used to represent the current value of the location counter.

### Local Symbols and Scoping Rules

Any identifier beginning with the dollar sign ($) is placed in the local symbol table instead of the general-purpose symbol table. The local symbol table can be purged with the .LOCAL directive. This allows simple scoping of identifiers, as shown below:

```
              .local                  ; new scope
$loop:        clr.b    (a0)+          ; clear a byte
clear_bytes:  dbra.w   d0,$loop       ; loop for the whole block
              rts                     ; return

              .local                  ; new scope
$loop:        move.b   (a0)+,(a1)+    ; copy a byte
copy_bytes:   dbra.w   d0,$loop       ; loop for the whole block
              rts                     ; return
```

In the above example, note that the symbol $LOOP is defined twice. The .LOCAL directive limits the scope of these labels to the code that references them. The symbols CLEAR_BYTES and COPY_BYTES are not local symbols because they do not begin with the dollar sign and are visible throughout the assembly unit. Local symbols cannot be made global or external.

### Symbol Resolution

All instruction mnemonics, register names, directives, macro names, and other permanent symbols are keywords that cannot be redefined by the user. These keywords are listed in Appendix III. The instruction:

```
move    equ     17
```

will result in an assembly error because the assembler treats this as a MOVE instruction and EQU is not appropriate as the source-effective address of a MOVE instruction. In fact, EQU is a keyword and the assembler will produce a message such as:

```
Syntax error:  Saw EQU but expected: <id>, (, #, ...)
```

## Expression Evaluation

All expressions are evaluated using 64-bit two's complement arithmetic. Intermediate results are stored in 32 bits. No overflow checking is performed, although division by 0 is detected and flagged as an assembler error. The result of evaluation is coerced into the result by removing the correct number of leading bits. Expressions fall into one of three categories:

- Absolute: These expressions contain only constants or symbols whose values are constant. Also, the difference of two relocatable symbols, both defined within the same section, is an absolute expression.

Simple relocatable: These expressions, although not constant, can be folded at assembly time. These include addition of a constant and a relocatable, or a relocatable minus an absolute.

Complex relocatable: These expressions, covering all of the remaining cases, include any expression that makes reference to an external symbol, the addition of two relocatable symbols, and so on.

If an expression cannot be evaluated because it is a complex relocatable expression, it is passed to the linker. All expressions are legal to the assembler. Some types of complex relocatable expressions have questionable meaning at best, and some are rejected by the linker and others by the target-specific loader. For example:

```
.ext.l  sym1,sym2

.sect   some_section,relocatable
sym3    equ     sym1**sym2
.end
```

If SYM1 and SYM2 are declared in another module as constants, the meaning of SYM3 is clear. If, however, SYM1 and SYM2 are relocatable entry points in a module, the meaning and value of SYM3 are unclear.

### Unary Operators

Unary operators +, –, and ~ are supported. The ~ operator produces the one's complement of its operand.

### Binary Operators

| | |
|---|---|
| = | Equality |
| /= | Inequality |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| * | Multiplication |
| / | Division |
| MOD | Modulo |
| REM | Remainder |
| & | Logical AND |
| ! | Logical inclusive OR |
| \ | Logical exclusive OR |
| + | Two's complement addition |
| – | Two's complement subtraction |
| >> | Shift right |
| << | Shift left |
| ** | Exponentiation |

### Operator Precedence

Operator precedence is, lowest to highest, as follows:

```
= /= < > <= >=
! \
&
+ -
* /
** >> <<
```

Within each line, the precedence of operators is the same.

# ASSEMBLER DIRECTIVES

## Listing Directives

The following listing directives control the content and format of the assembler listing:

| | |
|---|---|
| .LISTNC | List no conditionals |
| .LISTTC | List true conditionals only |
| .LISTC | List all conditionals (default) |
| .LISTMX | List macro expansion (default) |
| .LISTMC | List macro calls (default) |
| .LISTNM | List no macro expansions or calls |
| .LIST | Enable listing (default) |
| .NLIST | Disable listing |
| .TITLE | Specify the listing title |
| .SUBTTL | Specify the listing subtitle |
| .PAGE | Eject a page in the listing file |
| .BLANK | Place blank lines in the listing file |
| .HEAD | Define a header to be placed at the top of each subsequent page |
| .FOOT | Define a footer to be placed at the bottom of each subsequent page |
| .WIDTH | Define the width of the listing file |
| .LENGTH | Define the number of lines per listing page |

## Storage-Allocation Directives

The storage-allocation directives fall into three categories: uninitialized block storage, initialized unit storage, and initialized block storage. Each is described below.

### Uninitialized Block Storage

Each of these directives reserves storage for the specified number of elements by advancing the location counter as necessary.

| | |
|---|---|
| .DS.B | Reserve storage for bytes (8 bits) |
| .DS.W | Reserve storage for words (16 bits) |
| .DS.L | Reserve storage for longwords (32 bits) |
| .DS.S | Reserve storage for single-precision floating point (32 bits) |
| .DS.D | Reserve storage for double-precision floating point (64 bits) |

| | |
|---|---|
| .DS.X | Reserve storage for extended-precision floating point (96 bits) |
| .DS.A | Reserve storage for an address (32 bits) |

These directives are followed by an expression that represents the number of storage units to be reserved. For example:

```
.ds.w  10              ; allocate 10 words
```

### Initialized Unit Storage

These directives are used to allocate and initialize one or more units of storage. They are followed by a stream of values, which will be placed in consecutive locations within the current program section. For example:

```
.dc.w  10,0            ; allocate two words
                       ; set the first to 10
                       ; and the second to 0
```

| | |
|---|---|
| .DC.B | Define constant bytes (8 bits) |
| .DC.W | Define constant words (16 bits) |
| .DC.L | Define constant longwords (32 bits) |
| .DC.S | Define constant single-precision floating point (32 bits) |
| .DC.D | Define constant double-precision floating point (64 bits) |
| .DC.X | Define constant extended-precision floating point (96 bits) |
| .DC.A | Define constant addresses (32 bits) |
| .ASCII | Define a constant string, 8 bits per character |
| .ASCIZ | Define a constant text string, 8 bits per character, terminated with a null character |

### Initialized Block Storage

These directives are used to allocate a number of units and initialize them all to the same value. For example:

```
.dcb.w  10,0           ; allocate 10 words and
                       ; initialize them to zero
```

| | |
|---|---|
| .DCB.B | Define constant-block bytes (8 bits) |
| .DCB.W | Define constant-block words (16 bits) |
| .DCB.L | Define constant-block longwords (32 bits) |
| .DCB.S | Define constant-block single-precision floating point (32 bits) |
| .DCB.D | Define constant-block double-precision floating point (64 bits) |

| | |
|---|---|
| .DCB.X | Define constant-block extended-precision floating point (96 bits) |
| .DCB.A | Define constant-block addresses (32 bits) |

## Intermodule Symbol-Definition Directives

These directives inform the linker that the symbols specified are either defined in the current module for use in any module (global) or defined in another module and used by the current module (external). These directives can appear anywhere in the assembly module, either before or after the references to the symbols. For example:

```
.gbl.l  sym1,sym2,sym3
.ext.l  sym4,sym5,sym6
```

| | |
|---|---|
| .GBL.B | Byte global (8 bits) |
| .GBL.W | Word global (16 bits) |
| .GBL.L | Long global (32 bits) |
| .GBL.A | Address global (32 bits) |
| .EXT.B | Byte external (8 bits) |
| .EXT.W | Word external (16 bits) |
| .EXT.L | Long external (32 bits) |
| .EXT.A | Address external (32 bits) |

## Symbol-Definition Directives

The EQU instruction and any of several .DEF directives can be used to assign symbolic names to expressions. The .DEF directives allow the user to assign a value and a size attribute. Size attributes allow the assembler to generate optimal code even if the value of the symbol is not known.

Consider the following examples:

```
.ext.w  controller

.defp.w ctlr_reg1:=controller+3

move.b  d0,(controller+3)
move.b  d0,(ctlr_reg1)
```

Because CONTROLLER is external, its value is unknown. Although it has a 16-bit size, the expression CONTROLLER+3 has an unknown size. The first MOVE instruction will require four bytes for CONTROLLER+3 to ensure that the expression will fit at link time. The second MOVE instruction will allocate only two bytes for the value CTLR_REG1, and a link-time error will result if the actual value does not fit in two bytes. Note that this situation exists with

forward references or with expressions that contain forward references. Using these directives to specify the size of expressions that cannot be evaluated directly will produce smaller, faster code in processors that have multiple address representations, such as the M68000 family.

By contrast, the EQU instruction simply creates a symbol of implied size. For example:

```
ctlr_reg1   equ controller+3

move.b  d0,(controller+3)
move.b  d0,(ctlr_reg1)
```

These two MOVE instructions generate identical code. The symbol CTLR_REG1 is of use only to the programmer and may increase the readability of the source.

The symbol-definition directives include:

| | |
|---|---|
| .DEFP.B | Define a permanent byte symbol (8 bits) |
| .DEFP.W | Define a permanent word symbol (16 bits) |
| .DEFP.L | Define a permanent longword symbol (32 bits) |
| .DEFP.S | Define a permanent single-precision floating-point symbol |
| .DEFP.D | Define a permanent double-precision floating-point symbol |
| .DEFP.X | Define a permanent extended-precision floating-point symbol |
| .DEFT.B | Define a temporary byte symbol |
| .DEFT.W | Define a temporary word symbol |
| .DEFT.L | Define a temporary longword symbol |
| .DEFT.S | Define a temporary single-precision floating-point symbol |
| .DEFT.D | Define a temporary double-precision floating-point symbol |
| .DEFT.X | Define a temporary extended-precision floating-point symbol |
| SET | Define a temporary symbol |
| EQU | Define a permanent symbol |

Note that these directives can be used to create permanent or temporary symbols. Once defined, permanent symbols cannot be redefined. Temporary symbols can be redefined as frequently as desired if they are always defined with temporary type directives. A forward-referenced symbol cannot be defined subsequently as a temporary symbol. For example:

```
sym1 equ sym2+10
sym2 set 10
```

The attempt to define SYM2 as temporary is illegal here because, although it is previously undefined, there are forward references to it. Another example is:

```
sym2 set 0
syml equ sym2+10
sym2 set 10
```

In this example, SYM1 has a value of 10 throughout the assembly, because SYM2 was defined initially with a value of 0.

## Miscellaneous Directives

### CPU Directive

The CPU directive informs the assembler which CPU-family options are present in the specific target implementation. For example:

```
.CPU    "mc68020"
.CPU    "mc68881"
```

informs the assembler that the target processor is an MC68020 processor and has an MC68881 floating-point coprocessor. This directive causes any permanent symbols pertaining to the specified option to be placed in the permanent symbol table.

### SECT Directive

The .SECT directive is used to define, or switch between, program sections. When this directive is used to define a program section, it is followed by the name of the section and a list of parameters that describe the section. These parameters are:

- ABSOLUTE AT nnn: The section is absolute and starts at address *nnn*.
- RELOCATABLE: The section is relocatable.
- CODE: The section contains instructions. This attribute is meaningful only for processors that differentiate between instruction and data address spaces.
- DATA: The section contains constant data or variable data or both. This attribute is important only for processors that have physically distinct instruction and data address spaces.
- READWRITE: The section will be both read and written.
- READONLY: The section will be read only.
- OVERWRITE: Program sections from other modules with the same name as this section will be overwritten with data from this section.
- CONCATENATE: Program sections from other modules with the same name as this section will be concatenated at link time.
- ALIGNMENT := nnn: The alignment factor for this program section will be set to *nnn* bytes. See the .ALIGN directive for more detail.

When a .SECT directive is used to switch between sections, it is followed only by the section name. A section must be defined only once. Attempting to switch to a section that is undefined results in an error. Defining a section also switches to that section.

```
          .sect   prog,absolute at 16#1000#,code,readonly,alignment:=4
          .sect   heap,absolute at 16#4000#,data,readwrite,alignment:=1

          .sect   prog
          move.l  d0,temp
          .sect   heap
temp:     ds.l    1
```

The default parameters for a section are:

```
.sect   somename,relocatable,data,readwrite,concatenate,alignment:=2
```

### OFFSET Directive

The .OFFSET directive is much like the .SECT directive except that it changes the current section to be the NULL section and changes the current offset within the section to be a given constant. The .OFFSET directive normally is used to create mnemonic offsets into records or hardware registers. For example:

```
          .offset     0         ;   type info is
                                ;       record
name:     .ds.b       30        ;           name    : string (1..30);
age:      .ds.b       1         ;           age     : byte;
salary:   .ds.w       1         ;           salary  : integer;
info'size equ         .         ;       end;


          .sect   prog
update_age:                     ; pointer in a0, age in d0
          move.b  d0,(age,a0)
          rts

allocate_info:                  ; return pointer in a0
          move    #info'size,d0
          jsr     allocate
          rts
```

The .OFFSET directive allows any expression to be provided as the initial offset into the NULL section as long as the expression has no external or forward references. Attempts to generate code within the NULL section, with either assembler instructions or directives, are illegal.

### RADIX Directive

The .RADIX directive can be used to change both input and output radices. The input radix is used to process numeric literals encountered in the source. The output radix controls the listing

fields that display addresses and code generated by the assembler. The initial radix is decimal. Valid radices are 2, 8, 10, and 16.

```
.radix  10#16#      ; change the radix to HEX
.radix  2           ; change the radix to binary
```

Note that if the current radix is not clear, it is best to use based numeric constants to change radices. The base portion of a based numeric literal is always interpreted as decimal and is always unambiguous.

### IRADIX Directive

The .IRADIX directive is similar to the .RADIX directive, but it changes only the input radix.

### ORADIX Directive

The .ORADIX directive is similar to the .RADIX directive, but it changes only the output radix.

### REV Directive

The .REV directive accepts a character string that is placed in the object file produced by the assembler and is displayed by the linker in the link map. This is useful for tracking module revision level. For example:

```
.rev    "Version 2.3, last updated 7-aug-88"
```

### ALIGN Directive

The .ALIGN directive is used to realign the offset within the current section. The .ALIGN directive can be used in two forms. The first form requires an alignment factor. This alignment factor is an integer indicating a number of 8-bit bytes; the number of bits must be less than or equal to the alignment factor of the current section as specified by the .SECT directive. The second form of the .ALIGN directive has no alignment factor and uses the alignment factor of the current section. If the offset into the current section must be changed to ensure alignment, zeros are emitted into the current section until alignment is achieved. If the current section is the NULL section (see the .OFFSET directive), only the first form of the .ALIGN directive is allowed. Any positive, power of 2, alignment factor can be used within the NULL section. For example:

```
        .sect   heap,relocatable,data,readwrite,alignment:=4
        .sect   buffer,relocatable,data,readwrite,alignment:=1024
buffer1:.ds.b   100
        .align
buffer2:.sect   heap
block:  .ds.w   block'size*block_count
        .align  2
counter:.ds.w
        .align
```

In this example, BUFFER2 will begin 1,024 bytes after BUFFER1 because the .ALIGN directive realigned the section to the alignment factor given in the .SECT directive that defined the current section, BUFFER. The location COUNTER will be word-aligned regardless of the values of BLOCK'SIZE and BLOCK_COUNT. Also, the next storage allocated in section HEAP will be longword-aligned.

### OUTPUT Directive

The .OUTPUT directive allows the user's program to emit messages into the assembler listing and error-message file. Two forms of the directive exist: the first accepts a character string; the second accepts an arbitrary expression. Note that the expression's value must be static when the .OUTPUT directive is encountered; it cannot contain external or forward references. Each .OUTPUT directive produces a single line of text in the assembler output.

```
          .offset     0        ;   type info is
                               ;       record
name:     .ds.b       30       ;           name     : string (1..30);
age:      .ds.b       1        ;           age      : byte;
salary:   .ds.w       1        ;           salary   : integer;
info'size equ         .        ;       end;

          .output "The size of INFO is"
          .output info'size
```

### ERROR Directive

The .ERROR directive is like the first form of the .OUTPUT directive, except that it causes the semantic error count to be incremented. This causes the object module produced by the assembler to be marked as containing errors. Linking such modules produce warnings.

### INCLUDE Directive

The .INCLUDE directive can be used to cause a different file to be textually inserted into the source stream at the point of the directive. There is no limit to the number or nesting depth of .INCLUDE directives. The filename provided may employ Environment naming capabilities; names are resolved in the job's context. For example:

```
.include    "!users.wjh.project.data_definitions"
.include    "$'view(my_activity).units.macros"
```

## Repetitive Assembly

A looping primitive is provided to assist in creating complex tables, block structures, or instruction sequences that cannot be created with other directives and are too cumbersome to code manually. The looping construct causes a group of statements to be repeated as if they were actually duplicated within the source. For example:

```
.repeat 5
    move.b  (a0)+,(a1)+
.endrepeat
```

produces:

```
move.b      (a0)+,(a1)+
move.b      (a0)+,(a1)+
move.b      (a0)+,(a1)+
move.b      (a0)+,(a1)+
move.b      (a0)+,(a1)+
```

Although the repeat count shown here is a constant 5, the assembler allows any expression to be used as a repeat count if its value can be determined at the time the .REPEAT is encountered. This means that the expression cannot contain forward or external references. If the repeat count is less than 1, all text between the .REPEAT and .ENDREPEAT is ignored. There is no provision for creating unique labels within a repeat loop; if the application requires labels within a repeat loop, either use local symbols and place a .LOCAL directive inside the loop or use recursive macros. Placing nonlocal symbols within a repeat loop that is expanded more than once will result in errors because of multiply defined symbols. For example:

```
        count   set     1
        fact    set     1
        table_size equ 10
factorial_table:
        .repeat table_size
            .dc.l   fact
            count   set count + 1
            fact    set fact * count
        .endrepeat
```

This example creates a table that can be indexed by N to get N factorial.

## Conditional Assembly

The conditional-assembly primitive is an if-then-else construct that can be used to parameterize a single body of code to work under various circumstances. The following example depends on a symbol CPU to indicate whether an MC68020 instruction, CMP2, should be used or emulated:

```
mc68000 equ 68000
mc68010 equ 68010
mc68020 equ 68020
```

```
        cpu equ mc68010
        .if (cpu=mc68000)!(cpu=mc68010)      ;emulate if 68000 or 68010
            .local
            cmp.w   (a0),d0
            beq.s   $equal
            blt.s   $outofbounds
            cmp.w   (2,a0),d0
            beq.s   $equal
            blt.s   $outofbounds
            move.w  #2#00000#,ccr
            bra.s   $done
$equal:     move.w  #2#00100#,ccr
            bra.s   $done
$outofbounds:
            move.w  #2#0001#,ccr
$done:  .else                                ;don't emulate if 68020
            cmp2.w  (a0),d0
        .endif
```

In both the .REPEAT and .IF examples, instructions and directives were indented to clarify the structure. As always, white space is ignored by the assembler and can be used freely to meet various style requirements.

## Macro Assembly

The macro facility allows the programmer to define an identifier (a macro name) to be equivalent to a sequence of assembler statements. The definition is bounded by the .MACRO and .ENDMACRO directives.

When an identifier that is a macro name is encountered in the source stream (that is, in the sequence of text statements that is being assembled), the macro is expanded as follows:

- The remaining text on the source line is interpreted as the argument list for this invocation of the macro: it is scanned to determine the arguments. No identifiers in the line are expanded at this time; in particular, macro names encountered are not expanded.
- The text constituting the macro definition (that between the .MACRO and .ENDMACRO directives) is inserted into the source stream. Substitution for special symbols (including macro parameters, described below) occurs as they are encountered.
- Scanning of the source stream resumes at the location of the inserted macro text.

The macro definition can allow use of parameters that are included in the invocation. The supplied parameters are used within the sequence of statements according to the special symbols that represent them. The invocation also may specify a size qualifier appropriate to the target, such as L for longword.

Macro definitions can contain .IF and .REPEAT directives. These directives are processed when the macro is expanded. It is possible that the condition for the .IF is itself a parameter macro.

Within macro definitions there can be macro invocations that include additional macro invocations. However, macro definitions cannot be nested. The following symbols have special meanings within macro definitions:

%n Expands to the value of the *n*th parameter specified in the macro invocation; *n* represents one or more digits.

%% Expands to the data size specifier (for instance, W for word) specified in the macro invocation.

%: Expands to $*nnn*, where *nnn* is a sequence of digits unique to the macro definition. This can be used to generate unique labels with macros.

%# Expands to the number of parameters specified in the macro invocation.

%+ (Expands to nothing.) This is a concatenation operator for combining parameters and strings. For example, %1%+0 concatenates 0 to the value of the first parameter; %10 is the value of the tenth parameter.

The caret symbol (^) can precede a comma in the parameter list to indicate that the comma is part of the value being passed, not a value separator.

A macro is defined between .MACRO and .ENDMACRO directives. The .MACRO directive must have a macro name as its operand; for the .ENDMACRO directive, the name is optional. For example:

```
          .macro  mv            ; general-purpose move
          lea     (%1),%5       ; "%n" are parameter values
          lea     (%2),%6
loop%:    move.%% #%3,%4        ; "%:" for an identifier name
                                ; "%%" for size qualifier
          dbf     %4,loop%:
          .endmacro mv
```

A macro can be invoked any time after it is defined. Following is an example invocation and the resulting expansion:

```
          ; invocation
          mv.w    (Const^,PC), (Data^,A5), ; use ^ to quote commas
                  16, D0, A0, A1

          ; expansion is
          lea     (Const,PC),A0
          lea     (Data,A5),A1
loop$001: move.w #16,D0
          dbf     D0,loop$001
```

## CHARACTER USAGE

| | |
|---|---|
| A–Z | User symbol characters |
| a–z | User symbol characters |
| 0–9 | User symbol characters and numeric literals |
| ! | Logical inclusive OR operator |
| # | Used for immediate operands and based numeric literals |
| $ | User symbol character |
| % | Indicates special substitutions within macro expansion |
| & | Logical AND operator |
| * | Multiplication operator |
| ( ) | Used for operator precedence control and effective address syntax |
| - | Unary negation and two's complement subtraction operator |
| _ | User symbol character (underscore), ignored within numeric literals |
| + | Two's complement addition operator |
| = | Used for relational operators |
| [ ] | Used for effective address syntax |
| { } | Used for effective address syntax |
| : | Used for terminating labels |
| ; | Used for delimiting comments |
| ' | User symbol character (apostrophe) |
| " | Used with storage directives for creating ASCII strings and to provide string arguments to some directives |
| \ | Logical exclusive OR operator |
| \| | Line-continuation character |
| < | Used for relational operators |
| > | Used for relational operators |
| ? | User symbol character |
| , | Used as a separator (comma) |
| . | User symbol character (period) |
| ~ | One's complement unary operator |
| ^ | Used to pass commas as arguments to macro calls |
| ` | Not used, illegal (grave) |
| space | Separator |

| | |
|---|---|
| ascii.ht | Separator |
| ascii.lf | Statement separator |
| ascii.ff | Page delimiter |

# 6 MC68020/OS-2000 Cross-Linker

The MC68020/OS-2000 cross-linker is an important part of the Rational Cross-Development Facilities that typically runs automatically when a main unit is coded. The same linker is provided for all target computers; however, each target requires a different standard linker command file. The linker combines the contents of various object modules to produce an executable program. The modules can be named directly to the linker in the linker command file or indirectly via object-module libraries.

This chapter addresses individuals writing Ada and/or assembly-language programs or modules who need more explicit control or understanding of the linking process. The Cross-Development Facility was designed so that most users will use the default linking capability supplied during normal compilation; most users will never need to modify the standard linker command file. Some users may need to create their own command file, but the standard version can act as a starting point. Few, if any, users will ever need to invoke the linker explicitly. Instead, Rational intends the normal operation of the linker to be automatic when a main unit is promoted from the installed to coded state. The user should be familiar with assembly-language programming style and techniques, target-specific instructions and instruction syntax, and the Rational Cross-Development Facilities before using this material.

## TERMINOLOGY

The following terms are used in describing the linking process:

- Collection: A user-defined and user-named grouping of program sections that can be referenced as a single entity. The linker command file provides information about the number of collections, their object-module contents, and their names.
- Compilation unit: An Ada term that refers to an independently compilable Ada construct. A compilation unit can be a subprogram declaration or body, package declaration or body, generic declaration, or subunit.
- Linker command file: A text file used by the linker that contains commands specifying the following:
  - — Command filename
  - — Object modules to use as input for the linker

  - — Object-module libraries to use for additional input
  - — Collections to use during the linking process
  - — Memory segments to use for the linker output
  - — Attributes of the memory segments
  - — Additional commands for controlling content and placement of linker output
- Link map: A linker-generated text file that describes the linked, executable module.
- Memory segment: An address space provided by the target-computer architecture in which linker-processed code and/or data are stored. The linker command file provides information about the number of memory segments required, the collections to be placed in each memory segment, and the attributes of each memory segment.
- Object library: A grouping of object modules that are named as a single entity, from which the linker can select required modules. It is a file that contains a list of filenames.
- Object module: A binary file produced by an assembler that contains code, data, and relocation information for one or more program sections.
- Program section: A contiguous area of memory used to store the code for the program.
- Separate code and data: An architectural feature of some target computers in which the processor reads and writes data as though its address space were orthogonal to the address space from which it reads code. If the target processor supports this notion, it must be implemented in hardware to be effective.

## LINKER COMMAND (M68k.Link)

When you promote a main unit to the coded state with the Auto_Link switch set to true, the compilation system automatically invokes the linker to produce an executable module. The compilation system informs the linker of the object modules generated from the program's Ada units, so they can be linked. The user must name all assembly-language object modules for the linker—directly in the linker command file or indirectly via an object library.

The command to explicitly invoke the linker is:

```
M68k.Link (Command_File : String := "<IMAGE>";
           Exe_File : String := "<DEFAULT>";
           Map_File : String := "<DEFAULT>";
           Produce_Debug_Table : Boolean := False;
           Produce_Symbol_Table : Boolean := False;
           Produce_Symbol_Information : Boolean := False;
           Produce_Statistics : Boolean := False;
           Response : String := "<PROFILE>");
```

The parameters for this command are:

- `Command_File : String := "<IMAGE>";`

  Specifies the input file that contains linker commands. The default is the selected image.

- `Exe_File : String := "<DEFAULT>";`

  Specifies the executable file produced. The default appends _Exe to the value given to the Command_File parameter.

- `Map_File : String := "<DEFAULT>";`

  Specifies the file that will contain the link map. The default appends _Map to the value given to the Command_File parameter.

- `Produce_Debug_Table : Boolean := False;`

  Specifies whether a debug symbol table is generated. The default is false.

- `Produce_Symbol_Table : Boolean := False;`

  Specifies whether a symbol table file is generated. The default is false.

- `Produce_Symbol_Information : Boolean := False;`

  Specifies whether symbol cross-reference information will be added to the link map. The default is false.

- `Produce_Statistics : Boolean := False;`

  Specifies whether statistics of the assembly process are generated. The statistics will be found at the end of the link-map file. The default is false.

- `Response : String := "<PROFILE>";`

  Specifies how to respond to errors, how to generate logs, and what activities and switches to use during execution of this command. The default is the job response profile.

For example, assume that you want to link a file called User_Example with the files in the runtime library. You have created a linker command file called User_Linker_Command_File. You will use the default Exe, Map, Debug_Symbol, and Symbol_Table files. You want to generate a debug table but do not want any statistics. The following command accomplishes this:

```
M68k.Link (Command_File => "User_Linker_Command_File",
           Produce_Debug_Table => True);
```

# THE LINKING PROCESS

The following sections discuss internal events in the linking process. In actual operations, these events are invisible to the user. It is typically the case, however, that user-specified modules, libraries, and so on are specified even when using the supplied link stream. For extensive user-customization of the linker command file, more detailed knowledge of individual linker commands is required. See "Linker Command Files" in this chapter for more details.

## Loading the Specified Modules

During the first phase of linking, the linker locates and reads the object modules specified in the linker command stream. If any of the object modules cannot be read, this is noted in a message. Similarly, a message is output if a symbol is defined more than once.

## Scanning Object Libraries

Any remaining undefined symbols are resolved by scanning the specified object libraries. Each object module within a library is checked to see if it supplies a definition of an undefined symbol. If a module defines a needed symbol, that object module becomes part of the executable program. The linker scans the libraries in the order specified. Within each library, the object modules are searched in the order in which they appear. If undefined symbols remain after all libraries have been scanned, the symbols are reported and then defined by the linker as having the absolute value 0. This value may cause subsequent errors.

## Building Collections

The linker then segregates the program sections by grouping them into the collections defined by the user. A collection can consist of one or many program sections, but no two collections can contain the same program section. If any program sections are not named to be within any collection, they are reported and the linking process is aborted.

## Building Memory Segments

Some target computers have only one address space; others have many. Each address space for which the linker produces data is called a *segment*. The linker command file indicates the segments this program should have, the collections that should be placed in each segment, and the attributes of each segment (for example, read-only). The generation of output for a specific segment can be suppressed altogether; this feature can be useful for suppressing the data associated with uninitialized operand spaces in some computers. Every defined collection must be placed in a segment. Within each segment, memory is allocated for:

- Any program sections that were absolute at assembly time
- Collections that are bound to a specific location
- All remaining relocatable sections

If any assembly or link-time absolute sections overlap or fall outside the link-time-specified memory bounds of the segment, or if there is insufficient memory for the relocatable sections, appropriate messages are generated.

### Producing the Link Map

During the final stage, the linker produces the link map. The map can be used to determine where the linker placed certain program sections, the value of global symbols, the size of a program section or memory segment, where symbols are defined and used, and so on. For example, the link map contains the following:

- The names of all the segments specified in the linker command file
- The names of the object modules that each segment contains and the amount of unused memory
- The section to which each object module belongs, its starting address, and its word size in both hexadecimal and decimal notation
- A summary of all modules, including their filenames, their creation date and time, and their author
- A cross-reference of symbols known at link time (link-time symbols from object code generated from Ada units may not be recognizable)

## LINKER COMMAND FILES

The linker reads a text command file to determine the object modules and relocation methodology to be used in producing an executable program. Rational provides a standard linker command file that is used when the linker is invoked automatically. (See Appendix II for the complete pathname of this supplied default.) Names of libraries in the linker command file and names of modules within the libraries are resolved in their local context.

The CDF is designed so that most users can use the default linker command file during normal compilation. Many users will never need to modify the supplied linker command file. Some users may need their own command file, but they can use the standard version as a starting point because it is easily customizable. Although users can build their own linker command files, few will ever need to invoke the linker explicitly; instead, Rational intends the normal operation of the linker to be automatic. Users should follow the conventions described in the next section when writing their linker command files.

The name of a user-supplied linker command file must be supplied as the value of the Cross_Cg.Linker_Command_File library switch, described in "Cross-Compiler Switches" in Chapter 3. This name is resolved in the context of the containing switch file; it need not be a complete pathname. This makes it possible to employ the same linker command filename for different views of the same subsystem.

The following is an example of a linker command file:

```
program "Standard_OS2000_Linker_Commands" is

    use library "ada_runtime_library";
    use library "ada_runtime_library";

    collection shared_header           is (module$header);
    collection code                    is (ada_runtime_code, ada_code);
    collection constant_data           is (ada_runtime_const, ada_const);
    collection initial_values_header   is (module$initial_values_begin);
    collection initial_values          is (module$initial_values);
    collection initial_values_trailer  is (module$initial_values_end);
    collection shared_trailer          is (module$crc);

    collection unshared_header         is (module$writable_data_begin);
    collection initialized_data        is (module$initialized_data);
    collection writable_data           is (ada_runtime_data, ada_data);
    collection unshared_trailer        is (module$writable_data_end);

    segment shared is

        segment type is code;
        memory bounds are (0:16#fff_ffff#);

        place shared_header;
        place code;
        place constant_data;
        place initial_values_header;
        place initial_values;
        place initial_values_trailer;
        place shared_trailer;

    end;

    segment unshared is

        segment type is data;
        memory bounds are (0:16#fff_ffff#);

        place unshared_header;
        place initialized_data;
        place writable_data;
        place unshared_trailer;
        suppress;

    end;

end;
```

The supplied linker command text file is easily modified to include different module names or segment boundaries. Sophisticated user applications can include customized libraries and more complex collections. The linker command descriptions in the next section will assist advanced users who want to make more substantial changes to the linker command file.

The basic commands used in the linking process are described in the next section. They consist of reserved words, user-defined symbols, and strings.

The symbols can contain from 1 through 32 characters. The following characters may appear within the text of a symbol:

| | |
|---|---|
| A–Z | Letters of the alphabet (case-insensitive) |
| 0–9 | Decimal digits |
| _ | Underscore |
| . | Period |
| $ | Dollar sign |
| ' | Apostrophe |
| # | Pound sign; used in numerical expressions |

Strings are enclosed in double quotes ("").

The following reserved words have special significance in the linker command file. Reserved words must not be used as a user-specified name for object modules, library names, collection IDs, segment IDs, segment type IDs, or symbol IDs. Reserved words are case-insensitive.

| | | | |
|---|---|---|---|
| align | are | at | be |
| bounds | collection | end | exclude |
| force | is | libraries | library |
| link | memory | mod | others |
| place | program | resolve | section |
| segment | start | suppress | to |
| type | use | | |

## Basic Commands Used with Linker Command Files

Each linker command is described in detail in this section. The Backus-Naur form (BNF) definitions of linker command syntax are provided for reference in the following section.

Table 6-1 lists the linker commands and their purposes.

*Table 6-1 Linker Commands*

| Command | Purpose |
|---|---|
| Collection | Specifies what collections are to be created and what name is to be assigned to the collections. |
| Exclude | Specifies that the section is to be excluded. |
| Force | Specifies where in memory a given symbol is to be placed. |
| Link | Specifies what object modules are to be linked into the executable module. |
| Memory Bounds | Specifies what region of memory contains the specified segment. |
| Place | Specifies what collections are to be placed in the segment. |
| Program | Specifies the name of the linker command file and contains all the linker commands. |
| Resolve | Specifies where in memory a given symbol is to be placed. |
| Segment | Creates and names the segments that will receive data from the linker. |
| Segment Type | Specifies the user-defined segment type. |
| Start At | Specifies where the linker command file begins to write its data. |
| Suppress | Specifies that the segment is to be suppressed. |
| Use Library | Specifies what object libraries are scanned to resolve undefined symbols. |

The following notation is used to define the syntax for linker command files:

- The vertical bar ( | ) indicates that two symbols are alternatives. For example:

  ```
  lhs  --> AA | BB
  ```

  indicates that either symbols AA or BB are valid.

- Brackets ([ ]) indicate that the enclosed symbols are optional. For example:

  ```
  lhs  --> AA[,BB]
  ```

  indicates that either symbols AA or AA,BB are valid.

- Braces ({ }) indicate that the enclosed symbols can be repeated zero or more times. For example:

  ```
  lhs  --> AA{,BB}
  ```

  indicates that the symbols AA or AA,BB or AA,BB,BB or AA,BB,BB,BB and so on are valid.

### Program

The Program command is a block structure that specifies the name of the linker command file and contains all of the linker commands. This command is terminated with the reserved word *end*, followed by a semicolon.

The format of this command is:

**program** *file_name* **is** *linker commands* **end;**

The user-defined parameters of this command are:

- *file_name*: Specifies the filename associated with the linker command file. It is a string.
- *linker commands*: Specifies the particular commands that are executed.

An example of this command is:

```
program "linker_command_file_example" is
                .
                .
                .
        -- linker commands
                .
                .
                .
end;
```

### Link

The Link command specifies what object modules are to be linked into the executable module. This command is terminated by a semicolon.

The format of this command is:

**link** *file_name {, file_name}* ;

The user-defined parameter of this command is:

- *file_name*: Specifies the filename of the object module to be linked into the executable module. It is a string.

Examples of this command are:

```
link "module_a" , "module_b", "module_c";

link "module_a";
link "module_b";
link "module_c";
```

### Use Library

The Use Library command specifies what object libraries are scanned to resolve undefined symbols. This command is terminated by a semicolon.

The format of this command is:

```
use library | libraries file_name {, file_name } ;
```

The user-defined parameter of this command is:

- *file_name*: Specifies the filename of the object library to be scanned to resolve any undefined symbols. This is a string. Naming expressions may be used; they are resolved in the library or subsystem context of the linker command file.

Examples of this command are:

```
use libraries "ada_runtime_library", "example_library" ;

use library "ada_runtime_library" ;
use library "example_library" ;
```

### Collection

The Collection command specifies what collections are to be created and what name is to be assigned to the collections. This command is terminated by a semicolon.

The format of this command is:

```
collection collection_id is (section_name {, section_name}) ;
```

The user-defined parameters of this command are:

- *collection_id*: Specifies the user-defined name of the collection. The name can be 1 through 32 characters long (see the list of valid characters earlier in this section).
- *section_name*: Specifies the user-defined name of the section. Each section will contain one or more object modules. The section name and the object modules contained can be seen in the link map. The name can be 1 through 32 characters long.

A special form of this command uses the reserved name *others* to represent all sections that are not yet assigned to a collection:

```
collection collection_id is (others);
```

Examples of this command are:

```
collection a is (art_a, user_code_a) ;
collection b is (end_of_i) ;
collection c is (art_b, user_data_b) ;
collection d is (end_of_data) ;
```

The section order used in the collection command determines the order of program sections in the resulting collection. If more than one object module has placed data into a section, that data is ordered alphabetically by module name.

### Segment

The Segment command is used to create and name the memory segments that will receive data from the linker. This command is terminated by the reserved word *end* and a semicolon.

The format of this command is:

**segment** *segment_id* **is** *[segment type] [memory bounds] {place} [suppress segment]* **end** ;

The user-defined parameters of this command are:

- *segment_id*: Specifies the user-defined name given to the memory segment. The name can be 1 through 32 characters long (see above for valid characters).
- See the following sections for details about the subcommands Place, Memory Bounds, Segment Type, and Suppress Segment.

### Place

The Place command is a subcommand of the linker commands. It specifies what collections are to be placed in the segment. This subcommand is terminated by a semicolon.

The format of this subcommand is:

**place** *collection_id* ;

**place** *collection_id* **at** *expression* ;

**place** *collection_id* **align mod** *expression* ;

The user-defined parameters of this command are:

- *collection_id*: Specifies the user-defined name of the collection. The name can be 1 through 32 characters long (see above for valid characters).
- *expression*: Specifies a numerical memory address.

Examples of this subcommand are:

```
place data_collection ;

place data_collection at 16#0150# ;

place data_collection at 16#FFF_0150# ;

place data_collection align mod 16#0100# ;
```

### Memory Bounds

The Memory Bounds command is a subcommand of the linker commands. It specifies what region of memory contains the specified segment and is terminated by a semicolon.

The format of this subcommand is:

**memory bounds are** *(expression : expression) {expression : expression} ;*

The user-defined parameter of this subcommand is:

- *expression*: Specifies a numerical memory address. The first value is the beginning address and the second value is the ending address of the segment.

An example of this subcommand is:

```
memory bounds are (16#0100# : 16#FFF_FFFF#) ;
```

### Segment Type

The Segment Type command is an optional subcommand. It specifies the user-defined segment type. This subcommand is terminated by a semicolon.

The format of this subcommand is:

**segment type is** *segment_type_id ;*

The user-defined parameter of this subcommand is:

- *segment_type_id*: Specifies a user-defined value that typically identifies the type of collections found in the segment. It can be 1 through 32 characters long (see above for valid characters).

An example of this subcommand is:

```
segment type is linker_example ;
```

### Suppress Segment

The Suppress Segment command is a subcommand of the linker commands. It specifies that code or data associated with the segment are to be ignored. This subcommand is terminated by a semicolon.

The format of this subcommand is:

```
suppress ;
```

### Exclude Section

The Exclude Section command specifies what sections will be excluded from the executable module. This command is terminated by a semicolon.

The format of this command is:

**exclude section** *section_name ;*

The user-defined parameter of this command is:

- *section_name*: Specifies the name of the section that is to be excluded. It can be 1 through 32 characters long (see above for valid characters).

An example of this command is:

```
exclude section section_name ;
```

### Force or Resolve

The Force or Resolve command specifies the value to which a symbol will be resolved. The Resolve command will not redefine a symbol that is currently defined. This command is terminated by a semicolon.

The format of this command is:

**force | resolve** *symbol_name* sb_to be *expression ;*

The user-defined parameters of this command are:

- *symbol_name*: Specifies the name of the symbol that is to be placed at a particular memory location. It can be 1 through 32 characters long (see above for valid characters).
- *expression*: Specifies any valid value.

Examples of this command are:

```
resolve symbol_example to be 16#0199# ;

force symbol_example to be 57 ;
```

### Start At

The Start At command specifies the program counter where the program starts execution. This command is terminated with a semicolon.

The format of this command is:

**start at** *expression* ;

The user-defined parameter of this command is:

- *expression*: Specifies a numerical memory address.

Examples of this command are:

```
start at 16#0155# ;

start at 16#fff_0155# ;
```

# 7 Runtime Organization

## INTRODUCTION

This chapter describes the method by which the MC68020/OS-2000 Cross-Development Facility translates features of the Ada language onto the instruction-set architecture of the MC68020 and the facilities of the OS-2000 operating system. The topics discussed include memory organization, stack model, subprogram call and return sequences, parameter passing, exception handling, storage management, and tasking. The information in this document should be sufficient to enable a user to write assembly-language programs that interface with Ada programs or to modify a linker command file to achieve a desired program organization.

To make the best use of the information in this chapter, a reader should have knowledge of the Ada language, the M68000-family instruction set, the OS-2000 operating system, and techniques for mapping high-level languages onto computer architectures and operating systems.

## PROGRAM EXECUTION MODEL

The overall organization of a program, use of memory, and execution-time requirements constitute the program execution model. The compiler, linker command file, runtime system, target operating system, and target machine contribute to the definition of this model.

### Generated Code

The processing of a compilation unit generally results in the production of instruction sequences and the allocation of data storage. Allocated data may be either constant or modifiable and may be initialized or uninitialized. The term *generated code* describes all instructions and data produced by the compiler.

The compiler places the instructions, constant data, and modifiable data in three separate program sections named ADA_CODE, ADA_CONST, and ADA_DATA, respectively. The definition of the sections can be seen in the optional assembly or listing files produced by the compiler, as in the following directives:

```
.SECT    ADA_CODE,RELOCATABLE,CODE,READONLY,...
.SECT    ADA_CONST,RELOCATABLE,DATA,READONLY,...
.SECT    ADA_DATA,RELOCATABLE,DATA,READWRITE,...
```

Additional program sections for the OS-2000 runtime system include:

ADA_RUNTIME
ADA_RUNTIME_CONST
ADA_RUNTIME_DATA

The linker command file used for linking a main program specifies the placement of the program sections that constitute the program. The program sections required for an OS-2000 load module include:

MODULE$HEADER
MODULE$INITIAL_VALUES_BEGIN
MODULE$INITIAL_VALUES
MODULE$INITIAL_VALUES_END
MODULE$CRC
MODULE$WRITABLE_DATA_BEGIN
MODULE$WRITABLE_DATA
MODULE$WRITABLE_DATA_END

## Memory Usage

The generated code for an Ada program presumes no restrictions on the use of addresses within the M68K-family logical address space. As delivered by Rational, the linker command file restricts code and data addresses to be within 16#0FFF_FFFF# of the code loading address and the global database address, respectively.

## Processor Resource Utilization

This section discusses how the MC68020/OS-2000 runtime system uses registers and manages memory.

### Registers

The conventions observed by the Ada runtime model for the usage of the M68K-family registers are described below. Assembly-language or other subprograms that interface with Ada can assume that the conditions described hold upon entry and are required to satisfy the conditions before return.

- A7 is the stack pointer. Because trap handlers may run on the user stack, data above the top of the stack cannot be used reliably.
- A6 is the frame pointer. The structure of frames is described later in this document.

- A5 is the global data pointer. Because OS-2000 requires that data storage be position-independent, references to statically allocated data must be made indirect through a register. The Ada runtime model uses A5 for data indirection. (Note that many other OS-2000 programs, such as those produced by the OS-2000 C compiler, use A6 as the global data pointer.)
- A2 .. A4, D2 .. D7, and FP2 .. FP7 are nonvolatile registers. If the body of a subprogram uses any of these registers, their values must be saved on subprogram entry and restored before return to the calling environment. Conversely, a body of code that uses any of these registers can call to any subprogram and have the register values preserved across the call.
- A0, A1, D0, D1, FP0, and FP1 are volatile registers. The body of a subprogram can modify the values in these registers without saving the prior value. Conversely, if a body of code wants to preserve a value in one of these registers across a call, the value must be saved before the call and restored after the return.

### Memory-Management Options

The runtime system makes no presumption about the memory configuration of the execution hardware or about the location at which a program is loaded. No generated code or code in the runtime system references memory-management hardware.

## SUBPROGRAM CALL AND RETURN

The generated code for a call to a subprogram and the execution of the subprogram body generally result in the construction of a frame on the stack. The return from the subprogram and other generated code in the calling environment remove the frame. The frame consists of a number of words on the stack that contain the information required to perform parameter referencing, up-level referencing, exception handling, and subprogram return.

Figure 7-1 illustrates the general layout of information on the stack.

To better describe the use of the information contained in a frame, portions of the generated code for the following program fragment will be analyzed:

```
declare

    Local_Variable : Integer := 12;

    procedure Do_Something
                (Left : Integer; Right : Integer) is
    begin
        Local_Variable := Left + Right + Local_Variable;
    end Do_Something;

    function Compute_Result
                (Left : Integer; Right : Integer)
            return Integer is
```

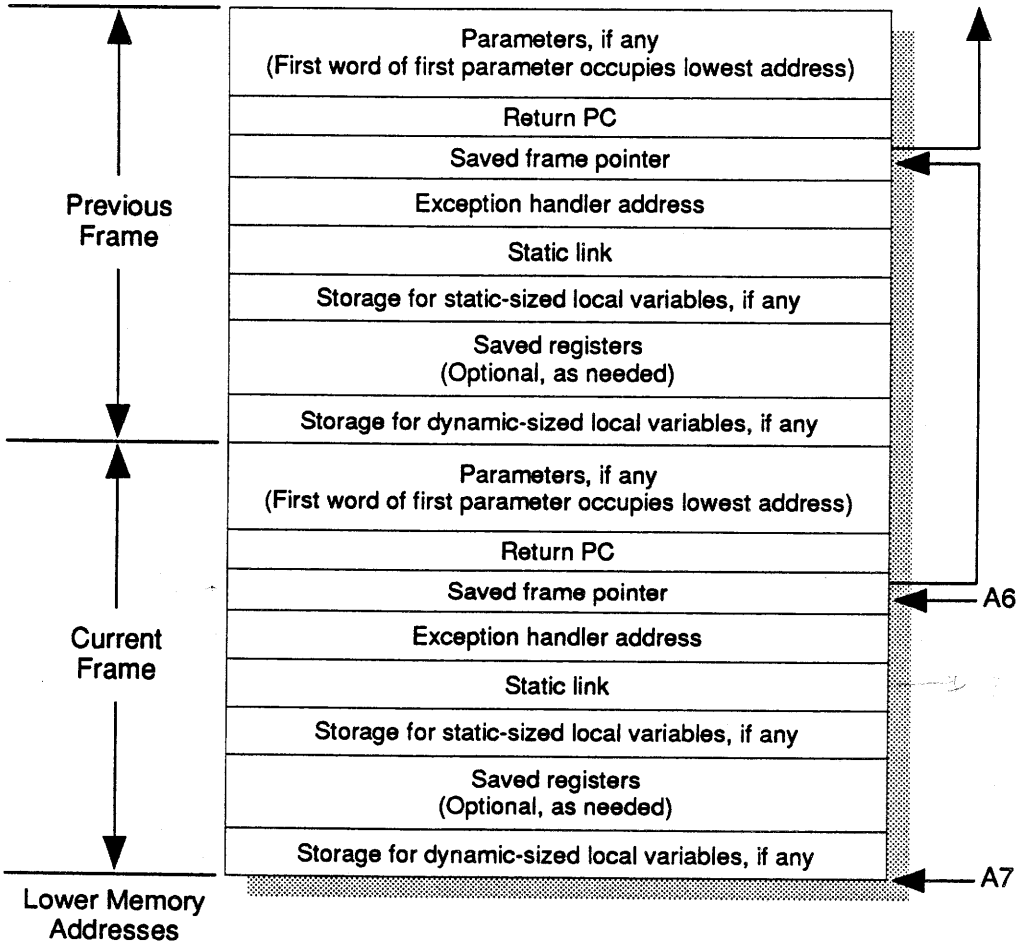



*Figure 7-1 Stack Model*

```
begin
    return Left + Right;
end Compute_Result;

begin
    Do_Something (Left => 9, Right => 3);
    Local_Variable := Compute_Result (Left => 12, Right => 9);
end;
```

It should be noted that the construction of frame information interacts strongly with code optimizations and that certain elements of a frame need not always be present. In the extreme, a source-language call may be expanded inline and result in no transfer of control. The following examples describe the general cases of call and return.

## A Simple Procedure Call

The first example is the procedure-call statement:

```
Do_Something (Left => 9, Right => 3);
```

The generated code for this call is:

```
MOVEQ     #3,D7                    ; note 1
MOVE.L    D7,-(A7)
MOVEQ     #9,D7                    ; 2
MOVE.L    D7,-(A7)
MOVEA.L   A6,A1                    ; 3
BSR.W     DO_SOMETHING             ; 4
ADDQ.W    #8,A7                    ; 5
```

These instructions perform the following operations:

1. The first pair of MOVEQ, MOVE.L instructions pushes the value 3 on the stack as the actual value for the Right formal parameter.
2. The second pair of MOVEQ, MOVE.L instructions pushes the value 9 on the stack as the actual value for the Left formal parameter.
3. The MOVEA.L instruction passes the current frame pointer in A1 for use as the static link.
4. The BSR.W instruction pushes the address of the instruction following the BSR instruction onto the stack and branches to the first instruction of the Do_Something procedure.
5. The ADDQ.W instruction pops from the stack the two words (8 bytes) that were allocated to pass the actual parameters.

The generated code for the body of Do_Something is:

```
DO_SOMETHING:
    LINK     A6,#0                 ;see following note 1
    CLR.L    -(A7)                                    ; 2
    MOVE.L   A1,-(A7)                                 ; 3

;statement # 1
    MOVE.L   (8,A6),D0                                ; 4
    ADD.L    (12,A6),D0                               ; 5
    TRAPV                                             ; 6
    ADD.L    D0,([-8,A6],-8)                          ; 7
    TRAPV                                             ; 8

Epilog:
    UNLK     A6                                       ; 9
    RTS                                               ;10
```

These instructions perform the following operations:

1. The LINK instruction pushes the value of the frame pointer on the stack and loads A0 with the address of the saved frame pointer.
2. The CLR instruction pushes the value 0 (zero) on the stack for the exception-handler address, indicating that there is no handler in this frame.
3. The MOVE instruction saves on the stack the value of the static link that was passed in A1.
4. This MOVE instruction loads D0 with the value of the Left formal parameter.
5. The ADD instruction computes the value of the subexpression (Left + Right).
6. The TRAPV instruction checks for numeric overflow, which might be caused by the previous arithmetic operation. If a trap occurs, the Numeric_Error exception will be raised.
7. The second ADD instruction computes the value of Left + Right + Local_Variable and stores the result in Local_Variable. Note that the indexing expression for Local_Variable is indirect and based on the static link stored at offset –8 in the frame.
8. The second TRAPV instruction checks for numeric overflow, which might be caused by the previous arithmetic operation.
9. The UNLK instruction pops the stack down to the saved frame pointer and then pops the value of the saved frame pointer in frame pointer A6. The return program counter remains at the top of the stack.
10. The RTS instruction pops an address from the top of the stack and branches to the instruction at that address.

## A Simple Function Call

The second example is the assignment statement in which the righthand side is a function call:

```
Local_Variable := Compute_Result (Left => 12, Right => 9);
```

The generated code for this statement is similar to that for the procedure call, except that after the two actual parameters are popped from the stack, the function result that is returned in D0 is stored into the location for Local_Variable.

The generated code for the body of the function is:

```
COMPUTE_RESULT:
        LINK    A6,#0                       ; note 1
        CLR.L   -(A7)
        PEA     (4)                             ; 2
        BSR.L   __STACK_CHECK
        LEA     (Handler,PC),A1                 ; 3
        MOVE.L  A1,(-4,A6)

; statement # 1
        MOVE.L  (8,A6),D0                       ; 4
        ADD.L   (12,A6),D0
        TRAPV
        BRA.B   Epilog

Handler:
        NOP                                     ; 5
        LEA     (Epilog,PC),A0                  ; 6
        MOVE.L  A0,(-4,A6)
        MOVE.L  D0,(-8,A6)                      ; 7
        LEA     (__Constraint_Error,A5),A0      ; 8
        CMPA.L  (-8,A6),A0
        BNE.L   __PROPAGATE_EXCEPTION

; statement # 2
        MOVEQ   #37,D0                          ; 9

Epilog:
        UNLK    A6                              ;10
        RTS
```

These instructions perform the following operations:

1. The LINK and CLR instructions of the prolog code set up the new frame pointer and set the exception-handler address to 0.
2. Four bytes of temporary local data storage will be allocated in the exception handler to contain the value of the active exception. The PEA and BSR instructions call the runtime stack-check routine to ensure that sufficient stack space is available; if it is not, the Storage_Error exception is raised.
3. The LEA and MOVE instructions save the address of the exception handler in the frame.
4. The code for computing the expression in the body of the function is similar to that for the procedure in the previous example, except that the computed expression is returned in D0.
5. Because there are no up-level references in the subprogram, the generated code for the exception handler begins with NOP to distinguish it from finalization code.

6. The LEA and MOVE instructions replace the exception-handler address with the address of the epilog code.
7. The next MOVE instruction saves the exception ID, which was in D0 upon entry into the handler in the frame.
8. The LEA, CMPA, and BNE instructions compare the raised exception with the exception ID for Constraint_Error and propagate the exception if the values are not equal.
9. The MOVEQ instruction places 37 into the function return register.
10. The epilog code for the function is the same as that for the procedure described previously.

## Parameter-Passing Conventions

Actual parameters to subprograms are passed on the stack. The environment of a call must push the correct number of parameters in the correct order before branching to a subprogram. The calling environment also must perform any required copy-back of parameters that are passed by value and remove the parameters from the stack after return from the call. The order in which parameters are passed is determined by the compiler and is subject to change.

The manner in which parameters are passed either in Ada subprograms or in other language subprograms that are to be interfaced with Ada can be specified by using the Export_Procedure, Export_Function, Import_Procedure, and Import_Function pragmas, as appropriate. Use of these and other pragmas is described in Appendix V.

The following paragraphs describe the conventions used for passing the actual parameters corresponding to various kinds of formal types.

### Scalar Types and Access Types

Objects of scalar and access types are passed by value on the stack. Scalar parameters occupy one or two longwords and are passed with the more significant bits in the lower memory address if two longwords are needed. Access parameters are passed as a single longword. The calling environment must perform copy-back associated with *out* and *in-out* parameters.

### Simple Record and Array Types

Simple record and array types include nondiscriminated record types, constrained subtypes of discriminated record types, and constrained array types and subtypes. Parameters of these simple types are passed by reference via a single longword on the stack containing the address of the object. The called subprogram must interpret the data at that address in a consistent manner.

If objects of a simple type are to be passed to assembly-language subprograms, representation specifications should be applied to the type to ensure a consistent interpretation of the object in the assembly code and in generated code. The compiler may choose a layout for objects whose type does not specify a representation. The manner in which the compiler chooses to represent objects is subject to change.

### Unconstrained Discriminated Record Types

Objects passed as actuals corresponding to formal parameters of unconstrained discriminated record types occupy one or two longwords on the stack. If the parameter is of mode *in*, a single longword containing the address of the object is passed. If the parameter is of mode *in out* or *out*, two longwords are passed. In the latter case, the longword at the higher memory address contains 0 if the actual is an unconstrained object or 1 if the actual is a constrained object. The longword at the lower memory address contains the address of the object.

### Unconstrained Array Types

Objects passed as actuals corresponding to formal parameters of unconstrained array types occupy two longwords on the stack. The longword at the higher memory address contains the address of a compiler-generated descriptor or dope vector for the array. The longword at the lower memory address contains the address for the data portion of the array object. The layout of the dope vector is subject to change, so we recommend that assembly-language subprograms not be written to manipulate objects of unconstrained array types.

### Functions Returning Scalar and Access Types

Scalar and access results from functions are returned in one or more registers. The registers used depend on the size and kind of the result, as shown in Table 7-1.

*Table 7-1 Function Return Conventions*

| Bit Size | Register | Comments |
|---|---|---|
| 32 | D0 | Enumeration types, integer types, fixed-point types, 32-bit floating-point types, access types |
| 64 | D0:D1 | 64-bit floating-point types |

### Functions Returning Simple Structures

Functions that return simple record and array types (as defined above) are converted by the compiler into procedures that have an additional *out* parameter of the same type as the function result. The function returns its result as would a procedure with this modified parameter profile.

### Functions Returning Unconstrained Structures

Several different mechanisms exist for returning unconstrained record or array function results. These mechanisims currently are not documented, and no attempts should be made to write assembly-language functions that return these objects. The current return mechanisms do not use the heap for their return values, which is an intentional compiler design goal.

## Finalization

Certain Ada-language features require that actions take place when leaving a block for any reason, including exception propagation. The code generated to perform these actions is called *finalization code.* Finalization code is generated to deallocate collections allocated within a block, await task termination, and terminate tasks.

## EXCEPTION HANDLING

Exception processing removes frames from the stack while searching for a handler for the exception being raised. Because an exception handler may depend on register contents to work correctly, the nonvolatile registers saved in each frame must be restored as the frames are removed. For this reason, each frame has an exception handler that is common with the return code for that frame. The exception processing assumes that A6 is a valid pointer to the current frame.

Because the call and return model ensures that the return program counter for a frame is at a static displacement from the frame pointer, it may be necessary for the runtime system to change the return program counter of the frame to point back into the runtime system. The runtime exception processor then jumps to the exception-handling code at the address pointed to by the exception handler within the frame, again at a static displacement from the frame pointer.

For most frames, this code is simply the epilog code for the subprogram, as outlined above for the simple procedure and simple function. This epilog code cuts back the stack, restores saved registers, and returns. Since the return program counter has been modified, control returns not to the caller of the subprogram but to the runtime system.

The runtime exception processor repeats this process until a frame with true exception-handling code is found. The runtime exception processor recognizes true exception-handling code by the initial NOP instruction, which distinguishes exception handlers from finalization code. The ID of the exception is available in D0 for the exception-handling code. Currently, an exception ID is the address of a constant string that is the fully qualified Ada name of the exception.

If an exception is propagated out of a procedure that is a main program, an error message will be output to Standard_Error, indicating that the program has terminated with an exception. Similarly, if an exception is propagated to the body of a task and not handled, a warning message will be output.

The runtime exception processing may be invoked in the following three ways:

- From generated code that corresponds either to an explicit raise statement or to an exception condition detected either dynamically or statically in the generated code
- From a machine trap handler that was entered by a hardware-detected exception condition
- From the runtime system when an exception condition is detected

## Exceptions Raised from Hardware Traps

Table 7-2 indicates the predefined exceptions that are raised as a consequence of MC68020 traps.

*Table 7-2 Exceptions Raised from Traps*

| Trap Name | Exception Name |
|---|---|
| CHK or CHK2 instruction | Constraint_Error |
| TRAPcc/TRAPV instruction | Numeric_Error |
| Zero divide | Numeric_Error |
| FPU zero divide | Numeric_Error |
| FPU operand error | Numeric_Error |
| FPU overflow | Numeric_Error |

## Exceptions Raised by the Runtime System

The following list indicates the situations in which exceptions are raised by the runtime system:

- Constraint_Error
  - — T'Image (X), where T is an enumeration type and X does not lie within T'First .. T'Last
  - — T'Pos (X), where T is an enumeration type with a representation clause and X does not lie within the range T'First .. T'Last
  - — T'Pred (X), where T is an enumeration type with a representation clause and X does not lie within the range T'First + 1 .. T'Last
  - — T'Succ (X), where T is an enumeration type with a representation clause and X does not lie within the range T'First .. T'Last – 1
  - — T'Value (S), where T is an enumeration type and the string S does not have the syntax of an enumeration literal or the enumeration literal specified by S does not exist for the base type T
  - — T'Width (X), where T is a subtype of an enumeration type that does not have static bounds and X does not lie in the range T'First .. T'Last
  - — T'Value (S), where T is an integer type and the string S does not have the synatx of a numeric literal or the numeric literal specified by S does not lie in the range Integer'First .. Integer'Last
- Program_Error
  - — Execution of a select statement that has no *else* part and for which all alternatives are closed (see LRM 9.7.1.11)

  - — Task elaboration error (see LRM 3.9.6)
- Storage_Error
  - — Attempts to allocate objects when there is insufficient storage within the collection and the collection cannot be extended
  - — Attempts to create access collections when there is insufficient storage within the heap
  - — Attempts to declare task objects when there is insufficient storage within the heap to allocate a task-control block
  - — Attempts to declare a task when there are insufficient system resources to create the message queues for the task
  - — Attempts to activate a task when there are insufficient system resources to create the stack for the task
  - — Attempts to execute a delay statement, timed entry call, or select with a delay alternative when there are insufficient system resources to create a timer for the required delay
  - — When the stack for an activation of a task or the main program becomes exhausted
- Tasking_Error
  - — At the end of a declarative part, when one or more task objects declared become completed during activation
  - — At the evaluation of an allocator, when one or more task objects created as components of the designated object become completed during activation
  - — Attempts to call an entry of a task that has completed its execution or becomes completed before accepting the call
  - — Attempts to call an entry of a task that is abnormal, becomes abnormal before accepting the call, or becomes abnormal during the rendezvous

## STORAGE MANAGEMENT

The storage manager provides support for dynamic memory allocation and deallocation associated with Ada access types.

### The Heap

The term *heap* refers to the memory from which collections, task-control blocks, and other runtime data structures are allocated. The memory to be used for the heap is acquired at program initialization by an F$SRqMem memory-request system call. The amount of memory to be requested is provided to the runtime system by generated code and can be set to a nondefault value by an argument to the Main pragma. If there are insufficient system resources to fulfill the memory request, a warning message is issued; any subsequent attempt to allocate storage from the heap will raise the Storage_Error exception.

## Collections

A *collection* is a data structure used by the runtime system to reserve a block of memory for allocation of objects of a given access type. (Note that this is not the same concept as the collection in a linker command file, described in the previous chapter.)

The runtime collection contains information to allow rapid reclamation of all associated memory when the given access type goes out of scope. In general, a collection is created for every access type at the point of elaboration of the access-type declaration. No collection will be created for an access type if a Storage_Size length clause is provided for the type with a value that is statically 0. In this case, no collection is allocated and any attempt to allocate or deallocate objects of this type will raise the Storage_Error exception.

There are two kinds of collections: extensible and nonextensible. The collection for an access type that has no Storage_Size length clause is extensible. Extensible collections are created with a default size determined by the runtime system. Allocations of objects from extensible collections will extend the collection automatically if there is insufficient free storage within the collection for the desired object. When a collection is extended, it is extended by the default size or the size of the object whose allocation necessitated the extension, whichever is larger. Nonextensible collections are created for collections that have an associated Storage_Size length clause. In this case, the collection is created with the size specified and will not be extended.

The Storage_Error exception may be raised by allocators that reference either type of collection.

### The Global Collection

The global collection is an extensible collection created at the earliest point of program elaboration and used to provide storage for dynamic-sized objects in static scopes. If no such objects exist, the global collection is not created. Note that if the collection is needed, it is created with a default size determined by the runtime system; the minimum size is 2K bytes. This storage then is inaccessible to the program throughout its execution.

### Dynamic Collections

Dynamic collections are created by the generated code at the point of access-type elaboration. Finalization code is generated to deallocate the collection when the scope in which the access type was declared is left. This happens at explicit block exit via a goto, exit, return, or end-of-block statement, as well as by leaving a block because of an exception. The latter case is handled by a compiler-generated finalization exception handler.

## Allocators

All allocations come from a collection—never directly from the heap. The compiler-generated size of the allocated object is rounded up to an even longword size. If the collection's free list does not contain a chunk of memory large enough and the collection is nonextensible or attempts to extend the collection fail, Storage_Error is raised. Allocation of objects with a size of 0 words are allocated one word (rounded up to two words) of storage to ensure that all allocations result in a unique object.

## Unchecked Deallocation

Unchecked deallocation is the only method of deallocating objects. The specified chunk of memory is added to the free list of the associated collection, and the free list is coalesced where possible.

# TASKING

This section discusses the problems of tasking and the runtime system.

## Tasks and Interprogram Communication

An Ada main program runs as a process in the OS-2000 system. Additionally, every task object in an Ada program is implemented as a separate OS-2000 process. The process corresponding to a task is initiated by an F$Fork system call, which allocates stack space for the process. Each task inherits four I/O paths from the process that runs as the main program: Standard_Input, Standard_Output, Standard_Error, and the file of error messages used by the runtime system.

For each task entry or member of a task-entry family, a message queue is created. One additional message queue is created per task for special use by the runtime system, as well as one message queue for the main program.

The runtime interfaces for associating and dissociating message queues for Interprogram Communication (IPC) are as specified in package Runtime_Support_For_Ipc:

```
with Message_Queue;

package Runtime_Support_For_Ipc is

    subtype Entry_Number is Natural range 1 .. 2 ** 15 - 1;

    type Status is (Successful, Invalid_Entry, Invalid_Queue);


    procedure Attach_Queue (For_Entry : Entry_Number;
                            New_Queue : Message_Queue.Id;
                            Result : out Status);
```

```
    -- Associate a new IPC message queue with the given entry of the
    -- task performing the call.  Any entry calls pending for the entry
    -- will result in Tasking_Error.
    -- Status is  Successful     - attach succeeded
    --            Invalid_Entry - current task has no entry with given
    --                            entry number
    --            Invalid_Queue - given queue id was not legal

    pragma Suppress (Elaboration_Check, Attach_Queue);
    pragma Interface (Asm, Attach_Queue);
    pragma Import_Procedure (Attach_Queue, "__ART_Attach_Queue",
                             Mechanism => (Value, Reference, Value));


    procedure Detach_Queue (For_Entry : Entry_Number; Result : out Status);
    -- Dissociate the current IPC queue from the given entry of the
    -- task performing the call and create a new regular Ada queue.
    -- Status is  Successful     - attach succeeded
    --            Invalid_Entry - current task has no entry with given
    --                            entry number
    --            Invalid_Queue - the queue for the given entry is not
    --                            an IPC queue

    pragma Suppress (Elaboration_Check, Detach_Queue);
    pragma Interface (Asm, Detach_Queue);
    pragma Import_Procedure (Detach_Queue, "__ART_Detach_Queue",
                             Mechanism => (Value, Value));

end Runtime_Support_For_Ipc;
```

The Storage_Error exception is raised at the point of declaration of a task object, if system resources are insufficient to fork the corresponding process or to create the required message queues.

When a task has terminated, the messages queues created for the task are deleted. The process corresponding to the task executes an F$Exit system call, thereby allowing the stack space that was allocated for the process to be reclaimed by the operating system.

See also Appendix V for details about compiler attributes associated with message queues and IPC.

## Priority

The priority of an Ada task can be specified by a Priority pragma in the task specification. The Ada priority of the main program is 127 by default; it can be changed in the declaration part with pragma Priority.

At program initiation, the runtime system queries the OS-2000 priority at which the program is executing. These two priority values determine a correspondence of Ada task priority to OS-2000 process priority, which is maintained when a task is created. For example, if a main program has priority 10 and is run at OS-2000 priority 100, a task within the program that has a specified priority of 20 will execute at priority 110 as an OS-2000 process.

## Timers

Three Ada-language constructs require timing to be performed: delay statements, timed entry calls, and selects with delay alternatives. When one of these statements is executed, a timer is started. If there are insufficient system resources to start the timer, Storage_Error is raised.

# 8 MC68020/OS-2000 Downloader

The MC68020 linker produces an executable module in the R1000 object-module format. Before this module can be used, however, its object-module format must be changed from R1000 to OS-2000 format. The MC68020/OS-2000 CDF provides the Convert command to change the formats. After the executable module has been converted to the appropriate object-module format, it must be downloaded to the target. The MC68020/OS-2000 CDF provides the Os2000_Put command to download the executable module. Once on the target, the file can be executed directly using the OS-2000 operating system commands. Alternatively, it can be executed through the use of the MC68020 cross-debugger (see Chapter 9).

## FORMAT-CONVERSION COMMAND (Convert)

The output of the MC68020 linker is an executable module in the R1000 object-module format. However, this format will not execute on the target. The format must first be converted to the OS-2000 object-module format before it can be executed.

The command that converts the formats is:

```
Convert (Old_Module : String  := "<IMAGE>";
         Old_Format : String  := "RATIONAL";
         New_Module : String;
         New_Format : String  := "<DEFAULT>";
         Options    : String  := "";
         Response   : String  := "<PROFILE>");
```

The parameters of this command are:

- `Old_Module : String := "<IMAGE>";`

  Specifies the name of the executable module that contains a non-OS-2000-compatible format.

- `Old_Format : String := "RATIONAL";`

  Specifies the object-module format of the old module. The default is Rational.

- `New_Module : String;`

  Specifies the name of the converted executable module.

- **`New_Format : String:= "<DEFAULT>";`**

  Specifies the object-module format of the new module. If the value of this string is "<DEFAULT>" and the enclosing world or subsystem has the Mc68020_Os2000 target key, then the new format is Mc68020_0s2000.

- **`Options : String:= " ";`**

  Presently no options are implemented.

- **`Response : String:= "<PROFILE>";`**

  Specifies how to respond to errors, how to generate logs, and what switches to use during execution of this command. The default is the job response profile.

## CONVERTING THE EXECUTABLE FILES

The executable file produced by the MC68020 linker is in the R1000 object-module format. To run on the target, it must be converted to the OS-2000 object-module format. To accomplish this, perform the following steps:

1. Create a Command window off the library that contains the executable module.
2. Enter **`Convert`** and press [Complete].
3. Enter the name of the executable module at the **`Old_Module`** prompt.
4. Enter the name of the executable module to be used on the target at the **`New_Module`** prompt.
5. Enter **`Mc68020_Os2000`** at the **`New_Format`** prompt.
6. Press [Promote] .

For example, the following command converts the object-module format:

```
Convert (Old_Module => "Main_68k.<exe>",
         Old_Format => "Rational",
         New_Module => "Main_68k",
         New_Format => "Mc68020_Os2000");
```

## TRANSFER COMMAND (Os2000_Put)

You must now transfer the executable module that has the OS-2000 object-module format to the target. The transfer command is:

```
Os2000_Put (From_Local_File : String := "<IMAGE>";
            To_Remote_File : String := "";
            Remote_Machine : String := "<DEFAULT>";
            Remote_Directory : String := "<DEFAULT>";
            Transliterate : Boolean := False;
            Response : Profile.Response_Profile := "<PROFILE>");
```

The parameters for this command are:

- **`From_Local_File : String := "<IMAGE>";`**

  Specifies the name of the file on the R1000 that contains the OS-2000-compatible executable module.

- **`To_Remote_File : String := "";`**

  Specifies the name of the executable module on the target. The default indicates that the name of the file on the R1000 will be used. If you do not use the same filename as the main unit on the R1000, you will not be able to debug your file.

- **`Remote_Machine : String := "<DEFAULT>";`**

  Specifies the name of the remote machine to which the executable module is transferred. The default uses the value of the Ftp_Profile.Remote_Machine switch in your switch file.

- **`Remote_Directory : String := "<DEFAULT>";`**

  Specifies the name of the remote directory that will receive the transferred executable module. The default uses the value of the Ftp_Profile.Remote_Directory switch in your switch file.

- **`Transliterate : Boolean := False;`**

  Specifies, when set to true, that the ascii.lf characters in the R1000 file be transmitted as ascii.cr. The default is false.

- **`Response : Profile.Response_Profile := "<PROFILE>";`**

  Specifies how to respond to errors, how to generate logs, and what switches to use during execution of this command. The default is the job response profile.

## TRANSFERRING THE EXECUTABLE FILES

The executable module is now in the OS-2000 object-module format, and you can transfer it to the target. To accomplish this, perform the following steps:

1. Create a Command window off the library containing the executable module.
2. Enter **`Os2000_Put`** and press [Complete].
3. Enter the name of the executable file on the R1000 at the **`From_Local_File`** prompt.
4. Enter the name of the executable file to be used on the target at the **`To_Remote_File`** prompt (if you want to debug this program later, it must have the same name as the program on the R1000).
5. Enter the name of the machine that will receive the executable module at the **`Remote_Machine`** prompt. (If you have set the Ftp_Profile.Remote_Machine switch in your switch file, you can use the default value for this parameter.)
6. Enter the name of the directory on the machine that will receive the executable module at the **`Remote_Directory`** prompt. (If you have set the Ftp_Profile.Remote_Directory switch in

your switch file, you can use the default value for this parameter.)

7. Press [Promote].

For example, the following command transfers the executable module Main_68k to the remote machine and directory specified in the switch file (the same name is retained, but ascii.lf characters are not transferred as ascii.cr):

```
Os2000_Put (From_Local_File => "Main_68k",
            To_Remote_File => "Main_68k",
            Remote_Machine => "<DEFAULT>",
            Remote_Directory => "<DEFAULT>",
            Transliterate => False,
            Response => "<PROFILE>");
```

## EXECUTING DIRECTLY ON THE TARGET

To run your executable module on the target, you enter the filename of the module from a console connected to the target. You can enter a full pathname or a simple filename if you are in the appropriate target directory.

The command is:

*executable_module_name* **-d -e -s**

The optional parameters are:

- *executable_module_name*: Specifies the filename of the executable module that you transferred to the target.
- **-d**: Specifies that task and elaboration diagnostics are to be run. This parameter is optional. If present, it must be separated from the executable module name and other parameters, if present, by a blank space.
- **-e**: Specifies that exception tracing should be done. With this option, whenever an exception is raised, the exception name and the location (in decimal) within the module in which it was raised are sent to Standard_Output.
- **-s**: Specifies that storage diagnostics are to be run. This parameter is optional. If present, it must be separated from the executable module name and other parameters, if present, by a blank space.

For example:

```
main_68k -e
```

executes the executable module found in Main_68k with exception tracing enabled.

# 9 MC68020/OS-2000 Cross-Debugger

The MC68020/OS-2000 Cross-Development Facility provides the ability to debug programs running on the target. Choosing the Mc68020_Os2000 target key selects the MC68020/OS-2000 cross-debugger. The same interface (!Commands.Debug) is used to control both the R1000 debugger and the MC68020/OS-2000 cross-debugger.

## DEBUGGER COMMANDS

Table 9-1 lists the commands in package !Commands.Debug that are used in debugging programs. These commands are used with both the R1000 native debugger and the MC68020/OS-2000 cross-debugger. For more information on these commands, consult the Debugging (DEB) book of the *Rational Environment Reference Manual.*

*Table 9-1 Package Debug Debugging Commands*

| Command | Function |
|---|---|
| Activate | Activates a previously removed (deactivated) breakpoint. |
| Address_To_Location | Displays the source location corresponding to the address of the specified machine instruction. |
| Break | Creates a breakpoint at the specified location in the specified task. |
| Catch | Stops execution whenever the named or selected exception is raised in the specified tasks at the specified location; reports the task name, the location in which the exception was raised, and the exception name. |
| Clear_Stepping | Removes all pending stepping operations that have been applied to the specified task(s). |
| Comment | Displays the comment specified by the string parameter in the Debugger window. |
| Context | Sets the specified context to be the specified pathname. |
| Convert | Converts the string specified in the number parameter to the specified base representation; 64-bit arithmetic is used. |

*Table 9-1 Package Debug Debugging Commands (continued)*

| Command | Function |
|---|---|
| Current_Debugger | Causes the named debugger to become the current default debugger for the user's session. |
| Debug_Off | Terminates debugging of the current job. |
| Disable | Enables or disables the option flag controlling the behavior of the debugger specified by the variable parameter. |
| Display | Displays an area of source in the Debugger window with statement numbers included, based on the current selection or the pathname provided. |
| Enable | Enables or disables the option flag controlling the behavior of the debugger specified by the variable parameter. |
| Exception_To_Name | Displays the source name of the exception that corresponds to the specified implementation-specific representation. |
| Execute | Commences (or resumes) execution of the named task(s). |
| Flag | Sets a flag controlling the behavior of the debugger to a specified string value. |
| Forget | Removes catch and propagate requests that match the Name, In_Task, and At_Location parameters. |
| History_Display | Displays a range of history entries for the specified task. |
| Hold | Stops execution of the specified task(s) and keeps it stopped until the task is explicitly released by the Release procedure or until an explicit request is given for execution of the task by an Execute or Run procedure. |
| Information | Lists information about the specified task. |
| Invoke | Starts the debugger on the selected main unit after determining the target key. |
| Kill | Kills the job being debugged and/or the debugger for the session. |
| Location_To_Address | Displays the code address for the machine instructions associated with the specified location. |
| Memory_Display | Displays the contents of memory. |
| Memory_Modify | Modifies the contents of memory. |
| Modify | Modifies or changes the values of the specified object. |
| Object_Location | Displays the machine address of the specified object (variable). |

*Table 9-1 Package Debug Debugging Commands (continued)*

| Command | Function |
|---|---|
| Propagate | Enters a request to the debugger that the program being debugged not be stopped when the specified exception is raised in the specified task at the specified location. |
| Put | Displays the value of the specified object in the Debugger window with formatting based on the type of the object. |
| Register_Display | Displays the registers for a given task and stack frame. |
| Register_Modify | Modifies the value of a register with a given hexadecimal value. |
| Release | Releases a task (or tasks) from the held state and moves the task to the stopped state. |
| Remove | Deactivates and possibly deletes the specified breakpoint(s). |
| Reset_Defaults | Resets all flag values and Boolean options to their standard values and unregisters all special displays. |
| Run | Executes the specified task(s) until the stop event has occurred the number of times specified by the Count parameter. |
| Set_Task_Name | Assigns a string *nickname* for the named task. |
| Set_Value | Sets the numeric variable flag controlling the behavior of the debugger to the specified value. |
| Show | Displays information about various debugger facilities. |
| Source | Finds the source for the specified location and displays that location highlighted in an Ada window. |
| Stack | Displays the specified frames of the stack of the named task. |
| Stop | Stops execution of the specified task(s). |
| Take_History | Enables or disables the recording of information about events executed in the specified task at a specified part of the program. |
| Target_Request | Issues a request to the target debug kernel. |
| Task_Display | Displays information about the named task(s). |
| Trace | Enables or disables the tracing of the specified events in the named task. |
| Trace_To_File | Sends trace output to the file specified by the File_Name parameter. |
| Xecute | Commences (or resumes) execution of the named task(s). |

Table 9-2 lists the commands in package !Commands.Common that are used in debugging programs. These commands are used with both the R1000 native debugger and the MC68020/OS-2000 cross-debugger. For more information on these commands, consult the Debugging (DEB) book of the *Rational Environment Reference Manual.*

*Table 9-2 Package Common Debugging Commands*

| Command | Function |
|---|---|
| Abandon | Deletes the Debugger window if the debugger has been killed; otherwise, the command has no effect. |
| Create_Command | Creates a Command window below the Debugger window if one does not exist; otherwise, the command puts the cursor in the existing Command window below the Debugger window. |
| Definition | Finds the defining occurrence of the designated element and brings up its image in a window on the screen. |
| Enclosing | Displays the library containing the Command window from which the job being debugged was started. |
| Release | Deletes the Debugger window if the debugger has been killed; otherwise, the command has no effect. |
| Write_File | Writes the current contents of the Debugger window into the named file. |
| Object.Child | Selects the Repeat child element of the currently selected element. |
| Object.First_Child | Selects the first child of the currently selected element. |
| Object.Last_Child | Selects the last child of the currently selected element. |
| Object.Next | Selects the Repeat next element past the currently selected element. |
| Object.Parent | Selects the parent element of the currently selected element. |
| Object.Previous | Selects the Repeat previous element before the currently selected element. |

## ADDITIONAL COMMANDS

For a full discussion on using the debugger, consult the Debugging (DEB) book of the *Rational Environment Reference Manual.* In addition to the commands described in DEB, some additional commands are available to support debugging at the machine level.

Table 9-3 lists the commands that are available for machine-level debugging. Hexadecimal (hex) numeric values in these commands must begin with a pound sign (#).

*Table 9-3 Machine-Level Debugging Commands*

| Command | Function |
|---|---|
| Debug.Address_To_Location | Displays the Ada source-code location of the specified address. |
| Debug.Invoke | Starts the debugger on the selected main unit after determining the target key. |
| Debug.Location_To_Address | Displays the address of the generated code for a selected source-code location. |
| Debug.Memory_Display | Displays the memory contents at a particular memory address. |
| Debug.Memory_Modify | Modifies up to a longword (32 bits) of memory. |
| Debug.Object_Location | Displays the machine address of the specified Ada unit (variable). |
| Debug.Register_Display | Displays the CPU registers for a given task and stack frame. |
| Debug.Register_Modify | Modifies the value of a register with a given hex value. |
| Debug.Run | Causes the debugger to step at the various levels supported. |

## Invoking the Debugger

There is no accelerated key binding for invoking the MC68020/OS-2000 cross-debugger. To start the MC68020/OS-2000 cross-debugger on a main program, select the unit and then execute the Debug.Invoke command. Typically, this is done in a Command window.

This command starts the debugger on the selected main unit after determining the target key. If a previous MC68020/OS-2000 debugger still exists, the default is to use that debugger rather than start a new one. Optionally, the name of the remote machine and the directory on the remote machine that contains the transferred executable module in the OS-2000 object-module format can be specified. If no values are specified, these values are determined from the Ftp.Remote_Directory and Ftp.Remote_Machine switches in the library switch file.

The format of the command is:

```
Debug.Invoke(Main_Unit => "<Image>",
             Options => "",
             Spawn_Job => True);
```

The parameters for this command are:

- **Main_Unit**: Specifies the name of the main program unit (the unit associated with pragma Main) that will be debugged. (The pragma may have been included in the unit specification or its body.)
- **Options**: Specifies the options to be used with the command. The possible options are:
  - **"Machine => Network Machine Name"**: Specifies the machine name of the remote computer.

— **"Directory => Target Directory Name"**: Specifies the name of the target directory that contains the executable module.

— **"Program => Target Program Name"**: Specifies the name of the executable module on the target. The program name or its OS-2000 process identification number (ID) must be specified if the target program name is different from the R1000 name. If the name applies to multiple processes on the target, then both the name and process ID must be specified. The target program can be a non-Ada program, but then debugging operations are limited.

— **"Process_ID => Target Process Id"**: Specifies the OS-2000 process ID of the module to be debugged. This unambiguously identifies the module. This parameter is meaningful only if Spawn_Job is set to false.

— **"User => User Login"**: Specifies the R1000 username to be identified with the debugging session. If this is omitted, the current user login name is assumed. The target-resident debugger code verifies that no other user has current debugging sessions on this target machine.

— **Reuse_Debugger**: Specifies a Boolean value that indicates whether the current debugger will be used. If set to false, a new debugger will be started and used.

- **Spawn_Job**: Specifies a Boolean value that indicates whether the current job is started by the debugger or is already running on the target. The default is true. If the value is false, an identifying program name or process ID must be specified in the Options parameter.

For example, from a world with target key Mc68020_Os2000 containing a coded main unit named Main_68k, whose executable has been converted and downloaded to the target where it has the same name, and with switch values that include:

```
    Debugging_Level := Full
Optimization_Level := 0
  Remote_Directory := "/h0/test"
    Remote_Machine := "Os2000_a"
```

Start a new debugger that initiates target program execution with the command:

```
Debug.Invoke(Main_Unit => "Main_68k",
             Options => "Reuse_Debugger => False");
```

## Determining Locations

The following commands are used to map from the Ada source to the machine representation of the code:

- Debug.Address_To_Location
- Debug.Location_To_Address
- Debug.Object_Location

### Debug.Address_To_Location

This procedure displays the Ada source-code location of the specified runtime address.

The format of this command is:

```
Debug.Address_To_Location(Address => "");
```

The parameter of this command is:

- **Address**: Specifies the memory address whose source location is to be determined. The address is a hexadecimal value—for example, #3A4B0FFF (up to 8 characters—32 bits).

For example, the following command selects the source-code location corresponding to the memory address #3A4B0FFF

```
Debug.Address_To_Location(Address => "#3A4B0FFF");
```

### Debug.Location_To_Address

This command displays the runtime address of the generated code for a selected source-code location.

The format of this command is:

```
Debug.Location_To_Address(Location => "<Selection>",
                          Stack_Frame => 0);
```

The parameters of this command are:

- **Location**: Specifies the selected source-code location.
- **Stack_Frame**: Specifies the stack frame. The default is 0, which means that the location parameter is used.

For example, the following command returns the address of the selected location:

```
Debug.Location_To_Address;
```

However, the following command displays the address of code for the current location indicated by frame 2 of the current task:

```
Debug.Location_To_Address("",2);
```

### Debug.Object_Location

This procedure displays the machine address of the specified object (variable).

The format of this command is:

```
Debug.Object_Location(Variable => "<Selection>",
                      Options => "");
```

The parameters of this command are:

- **Variable**: Specifies the object (variable) whose location is to be determined.
- **Options**: Specifies the options to be used with this command (no options are currently supported).

For example, the following command returns the location of the selected object:

```
Debug.Object_Location;
```

## Displaying Machine-Level Program Values

The following commands are used for displaying machine-level program values:

- Debug.Memory_Display
- Debug.Register_Display

### Debug.Memory_Display

This command displays the memory contents at a particular memory address.

The format of the command is:

```
Debug.Memory_Display(Address => "",
                     Count => 0,
                     Format => "Data");
```

The parameters of this command are:

- **Address**: Specifies the address at which to display memory. The address is a hexadecimal number (up to 8 characters—32 bits) or the name of a source location.
- **Count**: Specifies the number of locations (32-bit objects) to display.
- **Format**: Specifies the format of the data to be displayed. The Format parameter must specify either "Code" or "Data": "Code" disassembles memory as MC68020 instructions; "Data" displays memory in hexadecimal notation.

  When a name is given (pathname or <Selection>, <Cursor>, and so on), the address of the specified source is calculated and Count words are then displayed starting from that address and in the specified format.

For example, the following command displays 10 words of code starting at memory address #3A4B0FFF:

```
Debug.Memory_Display(Address => "#3A4B0FFF",
                     Count => 10,
                     Format => Code);
```

### Debug.Register_Display

This command displays the registers for a given task and stack frame.

The format of this command is:

```
Debug.Register_Display(Name => "",
                       For_Task => "",
                       Stack_Frame => 0,
                       Format => "");
```

The parameters for this command are:

- **Name**: Specifies the name of the registers to be displayed. If the null string ("") or "All" is specified, all CPU registers are displayed. (Consult *MC68020 32-Bit Microprocessor User's Manual*, 2nd ed., Prentice-Hall: Englewood Cliffs, New Jersey, 1985, for a listing and description of the registers.)
- **For_Task**: Specifies the task to be used. If the null string (""), the default, is specified, the current task is used.
- **Stack_Frame**: Specifies the stack frame to be used. If Stack_Frame = 0, the task is ignored and the physical machine registers are displayed. If the Stack_Frame > 0, the registers for the given task and the given frame are displayed.
- **Format**: Not used.

For example, the following command displays all of the register values:

```
Debug.Register_Display(Name => "All");
```

## Modifying Machine-Level Program Values

The following commands are used to modify memory or registers:

- Debug.Memory_Modify
- Debug.Register_Modify

### Debug.Memory_Modify

This command is used to modify up to a longword of memory (32 bits).

The format of this command is:

```
Debug.Memory_Modify(Address => ">>HEX ADDRESS<<",
                    Value => ">>HEX VALUE<<",
                    Width => 0,
                    Format => "Data");
```

The parameters of this command are:

- **Address**: Specifies the memory address to be modified. The address is a one- to eight-digit hexadecimal value (for example, #4A3B12AF).
- **Value**: Specifies the new value that is to be placed in the specified memory location. The value is also a one- to eight-digit hexadecimal value (for example, #FFFFFFFF).
- **Width**: Specifies the number of bits to be modified. The default value (0) represents the target storage-unit size, as specified in package System.
- **Format**: Not used.

For example, the following command places the value #FFFFFFFF in the memory location #4A3B12AF:

```
Debug.Memory_Modify(Address => "#4A3B12AF",
                    Value => "#FFFFFFFF"
                    Format => "Data");
```

### Debug.Register_Modify

This command is used to modify the value of a register with a given hexadecimal value.

The format of this command is:

```
Debug.Register_Modify(Name => ">>REGISTER NAME<<",
                      Value => ">>HEX VALUE<<",
                      For_Task => "",
                      Stack_Frame => 0,
                      Format => "");
```

The parameters of this command are:

- **Name**: Specifies the name of the register. (Consult *MC68020 32-Bit Microprocessor User's Manual*, 2nd ed., Prentice-Hall: Englewood Cliffs, New Jersey, 1985, for a listing and description of the registers.)
- **Value**: Specifies a one- to eight-digit hexadecimal value (for example, #3A4B12AF).
- **For_Task**: Specifies the name of the task (defaulted to the current task by the null string).
- **Stack_Frame**: Specifies an integer value (default 0). If Stack_Frame = 0, the task is ignored and the physical machine registers are modified. If the Stack_Frame > 0, the registers for the given task and the given frame are modified.
- **Format**: Not used.

For example, the following command places the value #FFFFFFFF in register D2:

```
Debug.Register_Modify(Name => "D2",
                      Value => "#FFFFFFFF"
                      Format => "");
```

## Program Control Commands

In addition to the foregoing commands, the following commands are used with the MC68020/OS-2000 cross-debugger:

- Debug.Break
- Debug.Current_Debugger
- Debug.Kill
- Debug.Run
- Debug.Target_Request

### Debug.Break

This command establishes a breakpoint at the specified memory address or Ada location.

The format of this command is:

```
Debug.Break(Location => "<Selection>",
            Stack_Frame => 0,
            Count => 1,
            In_Task => "",
            Default_Lifetime => True);
```

The parameters of this command are:

- **Location**: Specifies the location or memory address in hexadecimal notation at which the breakpoint is to be set.
- **Stack_Frame**: Specifies the frame in which to set the breakpoint.
- **Count**: Specifies the number of times the breakpoint must be executed before it interrupts the execution of the task.
- **In_Task**: Specifies the task in which the breakpoint is to be set.
- **Default_Lifetime**: Specifies whether the breakpoint is to be permanent or temporary.

For example, the following command sets a breakpoint at location #3A4B0FFF:

```
Debug.Break("#3A4B0FFF");
```

Using this command, it is possible to stop a program at a location in runtime code. Because the runtime code uses some nonstandard calling conventions to improve performance, the debug-

ger may be unable to display the stack or indicate the program location in this case. Instead, it reports that the program is stopped at an unknown location in the runtime system. Use the Run(Statement or Local_Statement) command to advance the program to the next Ada statement boundary.

### Debug.Current_Debugger

A given session may be running multiple debuggers of different target types—for example, an R1000 debugger and an MC68020/OS-2000 cross-debugger. When commands are entered into a new Debugger window, the new debugger becomes the current debugger. The command Debug.Current_Debugger also can designate one of these debuggers as the current debugger. All subsequent commands are directed to that debugger.

Running a debugger command in a window directly off an MC68020/OS-2000 cross-debugger window, or pressing a debugger key while in that window, makes the MC68020/OS-2000 cross-debugger the current debugger.

Running Debug.Invoke on an Mc68020_Os2000 unit also makes the MC68020/OS-2000 cross-debugger the current debugger.

The format of this command is:

```
Debug.Current_Debugger (Target => "");
```

The parameter of this command is:

- **Target**: Specifies the name of the cross-debugger.

For example, the following command sets the current debugger to be an MC68020/OS-2000 cross-debugger:

```
Debug.Current_Debugger(Target => "Mc68020");
```

### Debug.Kill

This command terminates the current debugging session and/or the target program. If the Debugger parameter is not set to true, the target program is terminated, but the debugger is still active and can be reused to debug another program.

The format of this command is:

```
Debug.Kill(Job => True,
           Debugger => True);
```

The parameters of this command are:

- **Job**: Specifies a Boolean value that determines whether the target program will be killed.
- **Debugger**: Specifies whether the debugger will be killed.

Using Debug.Kill to stop target-program execution is equivalent to issuing an abort request to the OS-2000. There is no guarantee that a blocked job will become unblocked to honor such a request.

Use Debug.Kill to stop execution of the debugger and avoid some processing overhead in the target. An active debugger monitors synchronous faults in the target even when no process is being debugged.

For example, the following command kills both the target job and the cross-debugger:

```
Debug.Kill(Job => True,
           Debugger => True);
```

### Debug.Run

This command can be used to provide Ada or machine-level stepping, depending on the setting of the Stop_At parameter. The Machine_Instruction value in the Stop_At parameter causes the debugger to step at the machine-instruction level.

The format of this command is:

```
Debug.Run(Stop_At => Debug.Statement,
          Count => 1,
          In_Task => "");
```

The parameters of this command are:

- **Stop_At**: Specifies the event that will cause the task to stop. The default is any statement.
- **Count**: Specifies the number of times the event must be executed before the task stops.
- **In_Task**: Specifies the task to be run.

For example, the following command causes the debugger to single-step at the machine level:

```
Debug.Run(Debug.Machine_Instruction);
```

Using this command, it is possible to stop a program at a location in runtime code. Because the runtime code uses some nonstandard calling conventions to improve performance, the debugger may be unable to display the stack or indicate the program location in this case. Instead, it reports that the program is stopped at an unknown location in the runtime system. Use the Run(Statement or Local_Statement) command to advance the program to the next Ada statement boundary.

### Debug.Target_Request

This command can be used to control some operations of the target-resident debugger. Among other things, it can set a stopping model for use with breakpoints. The value specified for the Options parameter determines which tasks on the target are halted when user breakpoint conditions are met.

The format of this command is:

```
Debug.Target_Request(Options := "",
                     In_File := "");
```

The In_File parameter is not currently used.

The Options parameter can be used in two modes: to control target operation via the debugger with keywords or to spawn a target program that can interact with the debugger. Possible Options keyword values are:

- **Stop Task**: Specifies that only the task whose breakpoint has been reached should stop.
- **Stop Program**: Specifies that all tasks associated with this debugging session (that is, all tasks associated with this program) should stop when the breakpoint is reached.
- **Stop Processor**: Specifies that all tasks on the target (except the debugger itself) should stop when the breakpoint is reached.
- **Trapon**: Specifies that tracing and stepping continue through the operating system TRAP instruction handler.
- **Trapoff**: Specifies that tracing and stepping operations in the debugger should ignore the operating system TRAP instruction handler, treating each call as a single instruction.

For example, the following command causes all tasks spawned by the executing program to stop when the next breakpoint is reached:

```
Debug.Target_Request (Options => "Stop Program");
```

The other mode for using the Options parameter can extend debugger functionality. In this mode, the debugger spawns a program that collects target-specific information and passes it to the user in the Debugger window. A decision about how to proceed with the debugging session can be made based on this information.

If the Options parameter string begins with a greater-than sign (>), the debugger interprets the remainder of the string as follows:

- If the next nonblank character is a percent sign (%), the hex digits immediately following are the name of an Ada program, which the spawned program (named in the next field) can use.
- The pathname of the OS-2000 program follows the > symbol and the task name field.
- After the pathname is an optional modifier field (maximum length 256 characters), which is passed to the spawned program in the Target_Request_Record.

The task name and modifier fields are optional. The debugger builds a target request record for communication with the program, spawns the program in the target with the OS-2000 call F$Fork, and then pauses ("sleeps"). The debugger passes the three standard I/O paths to the spawned program. There is no memory override for this spawned program: it is allocated only that memory specified in the OS-2000 module header. The debugger passes it a 10-character

command line, which is the memory address of the target request record: eight ASCII hexadecimal digits and two ASCII nulls.

It is the spawned program's responsibility to interpret the ASCII characters as an address, build a system pointer to access the target request record, and communicate with the debugger. A spawned program need not interact with the debugger. The debugger resumes operation after 30–60 seconds if the program terminates without communicating with the debugger.

The Ada code corresponding to the target request record and its representation clause are:

```
type Target_Request_Record is
    record
        Version : M68K_Word;
        Record_Address : Short_Array;
        Debugger_Pid : M68K_Word;
        Task_Pdsc : M68K_Address
        Task_Pdb : M68K_Address;
        System_Globals : M68K_Address;
        Debugger_Globals : M68K_Address;
        Task_Tcb : M68K_Address;
        Target_Request_Parameters : Long_Array;
        Nth_Call : M68K_Longword;
        Result_Ready : M68k_Byte;
        Filler : M68k_Byte;
        Last_Result : M68k_Byte;
        Severity : M68k_Byte;
        Result_Length : M68k_Word;
        Target_Request_Result : Long_Array;
    end record;

for Target_Request_Record use
    record
        Version at 0 range 0 .. 15;
        Record_Address at 2 range 0 .. 79;
        Debugger_Pid at 12 range 0 .. 15;
        Task_Pdsc at 14 range 0 .. 31;
        Task_Pdb at 18 range 0 .. 31;
        System_Globals at 22 range 0 .. 31;
        Debugger_Globals at 26 range 0 .. 31;
        Task_Tcb at 30 range 0 .. 31;
        Target_Request_Parameters at 34 range 0 .. 2047;
        Nth_Call at 290 range 0 .. 31;
        Result_Ready at 294 range 0 .. 7;
        Filler at 295 range 0 .. 7;
        Last_Result at 296 range 0 .. 7;
        Severity at 297 range 0 .. 7;
        Result_Length at 298 range 0 .. 15;
        Target_Request_Result at 300 range 0 .. 2047;
    end record;
```

The record fields are described in Table 9-4.

*Table 9-4 Target Request Record*

| Field | Contents |
|---|---|
| Version | Target request record version number; current value is 1 |
| Record_Address | A 10-character string; the command line passed by the debugger |
| Debugger_Pid | OS-2000 process ID of the spawning debugger |
| Task_Pdsc | A four-byte pointer to the OS-2000 process descriptor block; valid only if the *%name* parameter is used |
| Task_Pdb | A four-byte pointer to the debugger process database; not intended for customer use |
| System_Globals | The address of the OS-2000 system globals |
| Debugger_Globals | The address of the debugger globals; not intended for customer use |
| Task_Tcb | A four-byte pointer to the Ada task control block; valid only if the *%name* parameter is used |
| Target_Request_Parameters | A zero-terminated string; the modifier field of the Target_Request command; designed for direct use by the spawned program |
| Nth_Call | A counter indicating the number of times the program has returned results (useful when multiple values of the Target_Request_Result field must be passed); initially set to 1 and incremented by the debugger |
| Result_Ready | A byte that is initially 0; this is set to 1 by the spawned program to indicate that the Target_Request_Result field is ready for use by the debugger |
| Filler | An alignment byte |
| Last_Result | A byte that is initially 0; this should be set to 1 by the spawned program if the current Target_Request_Result is the last to be passed |
| Severity | An integer value set by the program to denote the result class: 0 for no message, 1 for information, 2 for a warning, 3 for an error, 4 for a fatal error, 5 for a catastrophic error |
| Result_Length | This integer value is set by the program to indicate the length of the string in the Target_Request_Result field |
| Target_Request_Result | A string returned by the program and passed by the debugger to the host; maximum length is 256 bytes; multiple partial results may be returned to build a larger string |

All fields are initially assigned values by the debugger. The spawned program should modify only the values of the Result_Ready, Last_Result, Severity, Result_Length, and Target_Request-_Result fields. The program then sends a wake-up signal to the target debugger using the process ID from the Debugger_Pid field. If this is not the last result, the spawned program pauses by executing a F$Sleep(0) system call. If this is the last result, the spawned program should terminate.

The debugger passes the Severity and Target_Request_Result values to the host. If this is not the last result (that is, if the program did not set the Last_Result field to 1), the debugger reinitializes the appropriate fields, passes control to the program, and pauses again. When the program completes, it sets the Last_Result field and all Target_Request_Result values are concatenated and displayed on the host in the Debugger window, with prefix characters determined by the Severity class.

## DIFFERENCES BETWEEN THE DEBUGGERS

The differences between the R1000 debugger and the MC68020/OS-2000 cross-debugger are discussed in this section.

### Breakpoints

On the MC68020, dead-code elimination results in the disappearance of statements. Breakpoints are refused at locations for which no code is generated.

Breakpoints can be set at machine addresses by specifying #<Address> for the location parameter.

Breakpoints can be set in specific instantiations of a generic but not in the generic itself.

### Exceptions

Unlike the native R1000 compiler, the MC68020/OS-2000 cross-compiler does not support flavors of exceptions—for example, additional detail such as "(Null Access)" when reporting "Constraint_Error".

The predefined exceptions (Constraint_Error, Storage_Error, Numeric_Error, and Program_Error) are always considered implicit. The R1000 debugger is able to distinguish between these when raised implicitly by the computer architecture or when raised via a raise statement. This distinction is not made in the MC68020/OS-2000 cross-debugger.

Exceptions in generics are specified by using their instantiation name.

When an exception is caught, the Ada location of the point of raise is correct. The program counter displayed (when the option address is true) is the program counter in the runtime system for exception processing.

The Information(Exceptions) command gives information for the most recently raised exception. Previous exceptions on the stack are not available. You cannot determine whether the last raised exception is still active. The raise location is not known unless you catch the exception in the debugger.

## Elaboration

To elaborate an M68000 program, a single task (the root task) is used to elaborate all the packages. Stepping and breakpointing operate on this task. Because this elaboration model differs from the one used on the R1000, stepping, breakpointing, and other operations that depend on task name behave differently during elaboration.

## Object Evaluation

No elaboration check is performed by the MC68020/OS-2000 debugger when it displays an object. Data displays before elaboration return whatever data are currently stored in that machine location.

Modification does not check that the value is in range. The debugger never corrupts adjacent data, but the value written may cause a subsequent reference to the modified object to get a constraint error. Array bounds are checked, since the debugger displays the value before modifying the value.

## Memory Display

The MC68020/OS-2000 cross-debugger supports two kinds of memory display: Data and Code. Data provides a hexadecimal dump and Code provides a disassembly listing. The R1000 also offers Control, Import, and Type, which are not supported on the MC68020/OS-2000 cross-debugger.

## Stack Frames

Block statements and accept statements are inlined by the MC68020/OS-2000 cross-compiler. They are displayed as separate frames (as on the R1000 debugger) even though no physical frame exists.

## Naming and Generics

The cross-debugger does not support naming of locations and exceptions by naming a generic unit; rather, the names must specify a particular instantiation. Similarly, the cross-debugger does not display the name of the generic along with the instantiation name. The explanations and examples to the contrary that are in the DEB volume of the *Rational Environment Reference Manual* apply to the native R1000 debugger, not to the cross-debugger.

## Target-System Characteristics

There are limits on use of the debugger with the MC68020/OS-2000 target. Only one user can debug a particular target system at a time, but that user can conduct multiple debugging sessions.

For a particular target, there can be no more than eight concurrent debugging sessions, no more than 64 processes being debugged on all sessions, and no more than 128 breakpoints for all processes and sessions.

The TCP/IP socket number that is used by the target-resident debugger to initiate a debugging session is configurable via a text file on the target.

# Appendix I: ASCII Table

*Standard 7-Bit ASCII Code*

| Bits B7, B6, and B5 are represented by the column headers. Bits B4, B3, B2, and B1 are represented by row headers. | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **000** | **001** | **010** | **011** | **100** | **101** | **110** | **111** |
| **0000** | *NUL* | *DLE* | *SP* | 0 | @ | P | ‘ | p |
| **0001** | *SOH* | *DC1* | ! | 1 | A | Q | a | q |
| **0010** | *STX* | *DC2* | " | 2 | B | R | b | r |
| **0011** | *ETX* | *DC3* | # | 3 | C | S | c | s |
| **0100** | *EOT* | *DC4* | $ | 4 | D | T | d | t |
| **0101** | *ENQ* | *NAK* | % | 5 | E | U | e | u |
| **0110** | *ACK* | *SYN* | & | 6 | F | V | f | v |
| **0111** | *BEL* | *ETB* | ’ | 7 | G | W | g | w |
| **1000** | *BS* | *CAN* | ( | 8 | H | X | h | x |
| **1001** | *HT* | *EM* | ) | 9 | I | Y | i | y |
| **1010** | *LF* | *SUB* | * | : | J | Z | j | z |
| **1011** | *VT* | *ESC* | + | ; | K | [ | k | { |
| **1100** | *FF* | *FS* | , | < | L | \ | l | \| |
| **1101** | *CR* | *GS* | - | = | M | ] | m | } |
| **1110** | *SO* | *RS* | . | > | N | ^ | n | ~ |
| **1111** | *SI* | *US* | / | ? | O | _ | o | *DEL* |

# Appendix II: Location of Components

This appendix indicates where important pieces of the MC68020/OS-2000 Cross-Development Facility can be found.

- Assembler, linker, and compiler

  The procedures to invoke the assembler and the linker and to restart the compiler can be found in:

  ```
  !Targets.Implementation.Motorola_68k'Spec_View.Units.M68k
  ```

- Runtime and default linker command files

  The runtime object-code files and the default linker command file can be found in:

  ```
  !Targets.Implementation.Mc68020_Os2000_Runtimes'View.Units
  ```

- Debugger

  The procedure for invoking the debugger can be found in:

  ```
  !Targets.Implementation.Mc68020_Os2000_Debuggers'Spec_View.Units
  ```

- Predefined I/O packages

  Packages Direct_Io, Sequential_Io, Text_Io, and Io_Exceptions can be found in:

  ```
  !Targets.Mc68020_Os2000.Io
  ```

- Other predefined packages

  Packages System, Calendar, Machine_Code, Unchecked_Conversion, and Unchecked-_Deallocation can be found in:

  ```
  !Targets.Mc68020_Os2000.Lrm
  ```

# Appendix III: Assembler and Linker Syntax

The following Backus-Naur form (BNF) is used to define the syntax for assembler and linker commands:

- Case: Uppercase text is used to denote terminal symbols; lowercase text is used to denote nonterminal symbols.
- | : The vertical bar indicates that two symbols are alternatives. For example:

  ```
  lhs  --> AA | BB
  ```

  indicates that either symbols AA or BB are valid.
- [ ]: Brackets indicate that the enclosed symbols are optional. For example:

  ```
  lhs  --> AA[,BB]
  ```

  indicates that either symbols AA or AA,BB are valid.
- { }: Braces indicate that the enclosed symbols can be repeated zero or more times. For example:

  ```
  lhs  --> AA{,BB}
  ```

  indicates that the symbols AA or AA,BB or AA,BB,BB or AA,BB,BB,BB and so on are valid.

## BNF FOR LINKER COMMAND FILES

The following BNF defines the structure of the contents of a linker command file:

```
command_file        ->  PROGRAM string IS linker_commands END ;

linker_commands     ->  { specify_modules }
                        { specify_libraries }
                        specify_collections { specify_collections }
                        specify_segments { specify_segments }
                        miscellaneous_cmds
```

```
specify_modules      ->  LINK string { , string } ;

specify_libraries    ->  USE LIBRARY | LIBRARIES string { , string } ;

specify_collections  ->  COLLECTION collection_id IS
                                ( section_name { , section_name } ) ;

specify_segments     ->  SEGMENT id IS segment_info END ;

segment_info         ->  [ memory_bounds ]
                     ->  { segment_type }
                     ->  { placement }
                     ->  [ suppress_segment ]

memory_bounds        ->  MEMORY BOUNDS ARE ( expression : expression );

segment_type         ->  SEGMENT TYPE IS id ;

placement            ->  PLACE collection_id ;
                     ->  PLACE collection_id AT expression ;
                     ->  PLACE collection_id ALIGN MOD expression ;

suppress_segment     ->  SUPPRESS

miscellaneous_cmds   ->  { set_symbol }
                     ->  [ set_start_pc ]

set_symbol           ->  RESOLVE | FORCE id TO BE expression ;

set_start_pc         ->  START AT expression


expression           ->  termp
                     ->  expression rel_op termp

termp                ->  termx
                     ->  termp or_ops termx

termx                ->  termz
                     ->  termx and_op termz

termz                ->  termq
                     ->  termz add_op termq

termq                ->  term
                     ->  termq mul_op term
```

```
term            ->  factor
                ->  term exp_op factor

factor          ->  element
                ->  |-| element
                ->  |+| element
                ->  NOT_OP element

element         ->  number
                ->  id
                ->  |(| expression |)|

rel_op          ->  |=|
                ->  |/=|
                ->  |>|
                ->  |<|
                ->  |>=|
                ->  |<=|

or_ops          ->  OR_OP
                ->  XOR_OP

and_op          ->  |&|

add_op          ->  |+|
                ->  |-|

mul_op          ->  |*|
                ->  |/|
                ->  MOD
                ->  REM

exp_op          ->  |**|
                ->  LSHIFT_OP
                ->  RSHIFT_OP
```

# BNF FOR ASSEMBLER COMMANDS

The following BNF is used with assembler commands.

## Target-Independent Syntax

```
module          --> statement_list end_statement <EOF>

statement_list  --> {[label] statement [comment] <EOL>}

label           --> symbol :

comment         --> ; {character}

statement       --> directive           |
                    instruction         |
                    conditional         |
                    repeat_statement    |
                    macro_call          |
                    macro_definition

instruction     --> (see target dependent syntax)

directive       --> listing                                     |
                    define_storage expression                   |
                    define_constant expression {, expression}   |
                    define_string string                        |
                    define_block expression , expression        |
                    define_globals symbol {, symbol}            |
                    define_externals symbol {, symbol}          |
                    define_permanent symbol := expression       |
                    define_temporary symbol := expression       |
                    symbol EQU expression                       |
                    symbol SET expression                       |
                    miscellaneous_directives                    |
                    output_directive                            |
                    section_directive                           |
                    align_directive

listing         --> .LISTNC          |   .LISTTC          |
                    .LISTC           |   .LISTMX          |
                    .LISTMC          |   .LISTNM          |
                    .LIST            |   .NLIST           |
                    .TITLE  string   |   .SUBTTL string   |
                    .PAGE            |   .BLANK           |
                    .HEAD            |   .FOOT
```

```
define_storage  --> .DS.B | .DS.W | .DS.L | .DS.S | .DS.D | .DS.X | .DS.A

define_constant --> .DC.B | .DC.W | .DC.L | .DC.S | .DC.D | .DC.X | .DC.A

define_string   --> .ASCII | .ASICZ

define_block    --> .DCB.B | .DCB.W | .DCB.L | .DCB.S |
                    .DCB.D | .DCB.X | .DCB.A

define_globals  --> .GBL.B | .GBL.W | .GBL.L

define_external --> .EXT.B | .EXT.W | .EXT.L

define_permanent--> .DEFP.B | .DEFP.W | .DEFP.L |
                    .DEFP.S | .DEFP.D | .DEFP.X

define_temporary--> .DEFT.B | .DEFT.W | .DEFT.L |
                    .DEFT.S | .DEFT.D | .DEFT.X
miscellaneous_directives--> .CPU string          |
                            .LOCAL               |
                            .PUSH expression     |
                            .RADIX expression    |
                            .IRADIX expression   |
                            .ORADIX expression   |
                            .INCLUDE string      |
                            .REV string          |
                            .ERROR string        |

output_directives--> .OUTPUT string      |
                     .OUTPUT expression

section_directives--> .SECT symbol {, section_param }     |
                      .OFFSET expression
```

```
section_param    --> ABSOLUTE AT expression      |
                     RELOCATABLE                 |
                     OVERWRITE                   |
                     CONCATENATE                 |
                     CODE                        |
                     DATA                        |
                     READONLY                    |
                     READWRITE                   |
                     ALIGNMENT := expression

align_directive --> .ALIGN [expression]

repeat_statement--> .REPEAT expression [comment] <EOL>
                        statement_list
                    .END REPEAT [comment] <EOL>
conditional      --> .IF expression [comment] <EOL>
                        statement_list
                    [.ELSE [comment] <EOL>
                        statement_list]
                    .ENDIF

macro_definition--> .MACRO macro_name [comment] <EOL>
                    statement_list
                    .ENDMACRO
macro_call       --> macro_name [actual_list]

actual_list      --> actual {,actual}

actual           --> expression | string

macro_name       --> symbol

end_statement    --> .END [comment] <EOL>

expression       --> factor                |
                     .IRADIX               |
                     .ORADIX               |
                     .POP                  |
                     .ALIGN                |
                     (expression)          |
                     unary_op expression   |
                     expression binary_op expression
```

```
unary_op        --> + | - | ~

binary_op       --> + | - | * | / | ** | | mod | rem |
                    ! | \ | & |
                    << | >> |
                    = | /= | > | < | >= | <=

factor          --> numeric_literal |
                    base_literal    |
                    char_literal    |
                    symbol

base_literal    --> numeric_literal # based_literal #

numeric_literal --> digit { _ | digit | alpha }

based_literal   --> alpha | digit { _ | digit | alpha }

symbol          --> local_symbol | regular_symbol

local_symbol    --> $ {alpha | digit | special_char }

regular_symbol  --> starting_char { alpha | digit | special_char }

starting_char   --> alpha | _ | .

digit           --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

alpha           --> a | b | c | d | e | f | g | h | i | j | k | l | m |
                    n | o | p | q | r | s | t | u | v | w | x | y | z |
                    A | B | C | D | E | F | G | H | I | J | K | L | M |
                    N | O | P | Q | R | S | T | U | V | W | X | Y | Z

special_char    --> $ | ' | _ | .

string          --> " {character} "

char_literal    --> ' character '
```

## M68000-Family Syntax

```
instruction     --> cpu_op | fpu_op | mmu_op

cpu_op  --> ABCD    efa_mode_00 , efa_mode_00
            ABCD    efa_mode_04 , efa_mode_04
            ADD     all_modes , efa_mode_00
            ADD     efa_mode_00 , alterable_memory_modes
            ADDA    all_modes , efa_mode_01
            ADDI    # immediate , alterable_data_modes
            ADDQ    # immediate_range_1_to_8 , alterable_modes
            ADDX    efa_mode_00 , efa_mode_00
            ADDX    efa_mode_04 , efa_mode_04
            AND     data_modes_1 , efa_mode_00
            AND     efa_mode_00 , alterable_memory_modes
            ANDI    # immediate , alterable_data_modes
            ANDI    # immediate , sr_or_ccr
            ASL     efa_mode_00 , efa_mode_00
            ASL     # immediate_range_1_to_8 , efa_mode_00
            ASL     alterable_memory_modes
            ASR     efa_mode_00 , efa_mode_00
            ASR     # immediate_range_1_to_8 , efa_mode_00
            ASR     alterable_memory_modes
            BCC     branch_displacement
            BCHG    efa_mode_00 , alterable_data_modes
            BCHG    bit_number , alterable_data_modes
            BCLR    efa_mode_00 , alterable_data_modes
            BCLR    bit_number , alterable_data_modes
            BCS     branch_displacement
            BEQ     branch_displacement
            BFCHG   dn_or_alterable_control_modes
            BFCLR   dn_or_alterable_control_modes
            BFEXTS  dn_or_control_modes , efa_mode_00
            BFEXTU  dn_or_control_modes , efa_mode_00
            BFFFO   dn_or_control_modes , efa_mode_00
            BFINS   efa_mode_00 , dn_or_alterable_control_modes
            BFSET   dn_or_alterable_control_modes
            BFTST   dn_or_control_modes
            BGE     branch_displacement
            BGT     branch_displacement
            BHI     branch_displacement
            BHS     branch_displacement
            BKPT    # immediate_range_0_7
            BLE     branch_displacement
            BLO     branch_displacement
            BLS     branch_displacement
            BLT     branch_displacement
            BMI     branch_displacement
            BNE     branch_displacement
```

```
BPL     branch_displacement
BRA     branch_displacement
BSET    efa_mode_00 , alterable_data_modes
BSET    bit_number , alterable_data_modes
BSR     branch_displacement
BTST    efa_mode_00 , data_modes_1
BTST    bit_number , data_modes_2
BVC     branch_displacement
BVS     branch_displacement
CALLM   # immediate_range_0_255 , control_modes
CAS     efa_mode_00 , efa_mode_00 , alterable_memory_modes
CAS2    register_pair , register_pair , cas2_efa
CHK     data_modes_1 , efa_mode_00
CHK2    control_modes , rn
CLR     alterable_data_modes
CMP     all_modes , efa_mode_00
CMP2    control_modes , rn
CMPA    all_modes , efa_mode_01
CMPI    # immediate , data_modes_2
CMPM    efa_mode_03 , efa_mode_03
DBCC    efa_mode_00 , dbcc_displacement
DBCS    efa_mode_00 , dbcc_displacement
DBEQ    efa_mode_00 , dbcc_displacement
DBF     efa_mode_00 , dbcc_displacement
DBGE    efa_mode_00 , dbcc_displacement
DBGT    efa_mode_00 , dbcc_displacement
DBHI    efa_mode_00 , dbcc_displacement
DBHS    efa_mode_00 , dbcc_displacement
DBLE    efa_mode_00 , dbcc_displacement
DBLO    efa_mode_00 , dbcc_displacement
DBLS    efa_mode_00 , dbcc_displacement
DBLT    efa_mode_00 , dbcc_displacement
DBMI    efa_mode_00 , dbcc_displacement
DBNE    efa_mode_00 , dbcc_displacement
DBPL    efa_mode_00 , dbcc_displacement
DBT     efa_mode_00 , dbcc_displacement
DBVC    efa_mode_00 , dbcc_displacement
DBVS    efa_mode_00 , dbcc_displacement
DBRA    efa_mode_00 , dbcc_displacement
DIVS    data_modes_1 , efa_mode_00
DIVS    data_modes_1 , register_pair
DIVSL   data_modes_1 , register_pair
DIVU    data_modes_1 , efa_mode_00
DIVU    data_modes_1 , register_pair
DIVUL   data_modes_1 , register_pair
EOR     efa_mode_00 , alterable_data_modes
EORI    # immediate , alterable_data_modes
EORI    # immediate , sr_or_ccr
EXG     rn , rn
```

```
EXT     efa_mode_00
EXTB    efa_mode_00
ILLEGAL
JMP     control_modes
JSR     control_modes
LEA     control_modes , efa_mode_01
LINK    efa_mode_01 , # immediate
LSL     efa_mode_00 , efa_mode_00
LSL     bit_number , efa_mode_00
LSL     alterable_memory_modes
LSR     efa_mode_00 , efa_mode_00
LSR     bit_number , efa_mode_00
LSR     alterable_memory_modes
MOVE    all_modes , alterable_data_modes
MOVE    sr_or_ccr , alterable_data_modes
MOVE    data_modes_1 , sr_or_ccr
MOVE    usp , efa_mode_01
MOVE    efa_mode_01 , usp
MOVEA   all_modes , efa_mode_01
MOVEC   control_register , rn
MOVEC   rn ,  control_register
MOVEM   register_list , movem_dest_mode
MOVEM   movem_src_mode , register_list
MOVEP   efa_mode_00 , efa_mode_05
MOVEP   efa_mode_05 , efa_mode_00
MOVEQ   # immediate_range_m128_to_127 , efa_mode_00
MOVES   rn , alterable_memory_modes
MOVES   alterable_memory_modes , rn
MULS    data_modes_1 , efa_mode_00
MULS    data_modes_1 , register_pair
MULU    data_modes_1 , efa_mode_00
MULU    data_modes_1 , register_pair
NBCD    alterable_data_modes
NEG     alterable_data_modes
NEGX    alterable_data_modes
NOP
NOT     alterable_data_modes
OR      data_modes_1 , efa_mode_00
OR      efa_mode_00 , alterable_memory_modes
ORI     # immediate , alterable_data_modes
ORI     # immediate , sr_or_ccr
PACK    efa_mode_04 , efa_mode_04 , # immediate_bit_16
PACK    efa_mode_00 , efa_mode_00 , # immediate_bit_16
PEA     control_modes
RESET
ROL     efa_mode_00 , efa_mode_00
ROL     # immediate_range_1_to_8 , efa_mode_00
ROL     alterable_memory_modes
ROR     efa_mode_00 , efa_mode_00
```

```
ROR     # immediate_range_1_to_8 , efa_mode_00
ROR     alterable_memory_modes
ROXL    efa_mode_00 , efa_mode_00
ROXL    # immediate_range_1_to_8 , efa_mode_00
ROXL    alterable_memory_modes
ROXR    efa_mode_00 , efa_mode_00
ROXR    # immediate_range_1_to_8 , efa_mode_00
ROXR    alterable_memory_modes
RTD     # immediate_range_m32768_to_32767
RTE
RTM     rn
RTR
RTS
SBCD    efa_mode_00 , efa_mode_00
SBCD    efa_mode_04 , efa_mode_04
SCC     alterable_data_modes
SCS     alterable_data_modes
SEQ     alterable_data_modes
SF      alterable_data_modes
SGE     alterable_data_modes
SGT     alterable_data_modes
SHI     alterable_data_modes
SHS     alterable_data_modes
SLE     alterable_data_modes
SLO     alterable_data_modes
SLS     alterable_data_modes
SLT     alterable_data_modes
SMI     alterable_data_modes
SNE     alterable_data_modes
SPL     alterable_data_modes
ST      alterable_data_modes
STOP    # immediate_bit_16
SUB     all_modes , efa_mode_00
SUB     efa_mode_00 , alterable_memory_modes
SUBA    all_modes , efa_mode_01
SUBI    # immediate , alterable_data_modes
SUBQ    # immediate_range_1_to_8 , alterable_modes
SUBX    efa_mode_00 , efa_mode_00
SUBX    efa_mode_04 , efa_mode_04
SVC     alterable_data_modes
SVS     alterable_data_modes
SWAP    efa_mode_00
TAS     alterable_data_modes
TRAP    # immediate_range_0_15
TRAPCC
TRAPCC  # immediate
TRAPCS
TRAPCS  # immediate
TRAPEQ
```

```
                TRAPEQ  # immediate
                TRAPF
                TRAPF   # immediate
                TRAPGE
                TRAPGE  # immediate
                TRAPGT
                TRAPGT  # immediate
                TRAPHI
                TRAPHI  # immediate
                TRAPHS
                TRAPHS  # immediate
                TRAPLE
                TRAPLE  # immediate
                TRAPLO
                TRAPLO  # immediate
                TRAPLS
                TRAPLS  # immediate
                TRAPLT
                TRAPLT  # immediate
                TRAPMI
                TRAPMI  # immediate
                TRAPNE
                TRAPNE  # immediate
                TRAPPL
                TRAPPL  # immediate
                TRAPT
                TRAPT   # immediate
                TRAPV
                TRAPVC
                TRAPVC  # immediate
                TRAPVS
                TRAPVS  # immediate
                TST     data_modes_2
                UNLK    efa_mode_01
                UNPK    efa_mode_04 , efa_mode_04 , # immediate_bit_16
                UNPK    efa_mode_00 , efa_mode_00 , # immediate_bit_16

mmu_op  -->     PBBS    branch_displacement
                PBLS    branch_displacement
                PBSS    branch_displacement
                PBAS    branch_displacement
                PBWS    branch_displacement
                PBIS    branch_displacement
                PBGS    branch_displacement
                PBCS    branch_displacement
                PBBC    branch_displacement
                PBLC    branch_displacement
                PBSC    branch_displacement
                PBAC    branch_displacement
```

```
PBWC    branch_displacement
PBIC    branch_displacement
PBGC    branch_displacement
PBCC    branch_displacement
PDBBS   efa_mode_00 , dbcc_displacement
PDBLS   efa_mode_00 , dbcc_displacement
PDBSS   efa_mode_00 , dbcc_displacement
PDBAS   efa_mode_00 , dbcc_displacement
PDBWS   efa_mode_00 , dbcc_displacement
PDBIS   efa_mode_00 , dbcc_displacement
PDBGS   efa_mode_00 , dbcc_displacement
PDBCS   efa_mode_00 , dbcc_displacement
PDBBC   efa_mode_00 , dbcc_displacement
PDBLC   efa_mode_00 , dbcc_displacement
PDBSC   efa_mode_00 , dbcc_displacement
PDBAC   efa_mode_00 , dbcc_displacement
PDBWC   efa_mode_00 , dbcc_displacement
PDBIC   efa_mode_00 , dbcc_displacement
PDBGC   efa_mode_00 , dbcc_displacement
PDBCC   efa_mode_00 , dbcc_displacement
PFLUSH  mmufc , bit_4
PFLUSH  mmufc , bit_4 , alterable_control_modes
PFLUSHA
PFLUSHS mmufc , bit_4
PFLUSHS mmufc ,  bit_4 , alterable_control_modes
PFLUSHR memory_modes
PLOADR  mmufc , alterable_control_modes
PLOADW  mmufc ,alterable_control_modes
PMOVE   mmu_reg , alterable_modes
PMOVE   all_modes , mmu_reg
PRESTORE movem_src_mode
PSAVE   movem_dest_mode
PSBS    alterable_data_modes
PSLS    alterable_data_modes
PSSS    alterable_data_modes
PSAS    alterable_data_modes
PSWS    alterable_data_modes
PSIS    alterable_data_modes
PSGS    alterable_data_modes
PSCS    alterable_data_modes
PSBC    alterable_data_modes
PSLC    alterable_data_modes
PSSC    alterable_data_modes
PSAC    alterable_data_modes
PSWC    alterable_data_modes
PSIC    alterable_data_modes
PSGC    alterable_data_modes
PSCC    alterable_data_modes
PTESTR  mmufc , alterable_control_modes , bit_3
```

```
                PTESTR  mmufc , alterable_control_modes , bit_3,efa_mode_01
                PTESTW  mmufc , alterable_control_modes , bit_3
                PTESTW  mmufc , alterable_control_modes , bit_3,efa_mode_01
                PTRAPBS
                PTRAPBS # immediate
                PTRAPLS
                PTRAPLS # immediate
                PTRAPSS
                PTRAPSS # immediate
                PTRAPAS
                PTRAPAS # immediate
                PTRAPWS
                PTRAPWS # immediate
                PTRAPIS
                PTRAPIS # immediate
                PTRAPGS
                PTRAPGS # immediate
                PTRAPCS
                PTRAPCS # immediate
                PTRAPBC
                PTRAPBC # immediate
                PTRAPLC
                PTRAPLC # immediate
                PTRAPSC
                PTRAPSC # immediate
                PTRAPAC
                PTRAPAC # immediate
                PTRAPWC
                PTRAPWC # immediate
                PTRAPIC
                PTRAPIC # immediate
                PTRAPGC
                PTRAPGC # immediate
                PTRAPCC
                PTRAPCC # immediate
                PVALID  VAL , alterable_control_modes
                PVALID  efa_mode_01 , alterable_control_modes

fpu_op  -->     FABS    data_modes_1 , fpn
                FABS    fpn , fpn
                FABS    fpn
                FACOS   data_modes_1 , fpn
                FACOS   fpn , fpn
                FACOS   fpn
                FADD    data_modes_1 , fpn
                FADD    fpn , fpn
                FASIN   data_modes_1 , fpn
                FASIN   fpn , fpn
                FASIN   fpn
```

```
FATAN   data_modes_1 , fpn
FATAN   fpn , fpn
FATAN   fpn
FATANH  data_modes_1 , fpn
FATANH  fpn , fpn
FATANH  fpn
FBF     branch_displacement
FBEQ    branch_displacement
FBOGT   branch_displacement
FBOGE   branch_displacement
FBOLT   branch_displacement
FBOLE   branch_displacement
FBOGL   branch_displacement
FBOR    branch_displacement
FBON    branch_displacement
FBUEQ   branch_displacement
FBUGT   branch_displacement
FBUGE   branch_displacement
FBULT   branch_displacement
FBULE   branch_displacement
FBNE    branch_displacement
FBT     branch_displacement
FBSF    branch_displacement
FBSEQ   branch_displacement
FBGT    branch_displacement
FBGE    branch_displacement
FBLT    branch_displacement
FBLE    branch_displacement
FBGL    branch_displacement
FBGLE   branch_displacement
FBNGLE  branch_displacement
FBNGL   branch_displacement
FBNLE   branch_displacement
FBNLT   branch_displacement
FBNGE   branch_displacement
FBNGT   branch_displacement
FBSNE   branch_displacement
FBST    branch_displacement
FCMP    data_modes_1 , fpn
FCMP    fpn , fpn
FCOS    data_modes_1 , fpn
FCOS    fpn , fpn
FCOS    fpn
FCOSH   data_modes_1 , fpn
FCOSH   fpn , fpn
FCOSH   fpn
FDBF    efa_mode_00 , dbcc_displacement
FDBEQ   efa_mode_00 , dbcc_displacement
FDBOGT  efa_mode_00 , dbcc_displacement
```

```
FDBOGE   efa_mode_00 , dbcc_displacement
FDBOLT   efa_mode_00 , dbcc_displacement
FDBOLE   efa_mode_00 , dbcc_displacement
FDBOGL   efa_mode_00 , dbcc_displacement
FDBOR    efa_mode_00 , dbcc_displacement
FDBON    efa_mode_00 , dbcc_displacement
FDBUEQ   efa_mode_00 , dbcc_displacement
FDBUGT   efa_mode_00 , dbcc_displacement
FDBUGE   efa_mode_00 , dbcc_displacement
FDBULT   efa_mode_00 , dbcc_displacement
FDBULE   efa_mode_00 , dbcc_displacement
FDBNE    efa_mode_00 , dbcc_displacement
FDBT     efa_mode_00 , dbcc_displacement
FDBSF    efa_mode_00 , dbcc_displacement
FDBSEQ   efa_mode_00 , dbcc_displacement
FDBGT    efa_mode_00 , dbcc_displacement
FDBGE    efa_mode_00 , dbcc_displacement
FDBLT    efa_mode_00 , dbcc_displacement
FDBLE    efa_mode_00 , dbcc_displacement
FDBGL    efa_mode_00 , dbcc_displacement
FDBGLE   efa_mode_00 , dbcc_displacement
FDBNGLE  efa_mode_00 , dbcc_displacement
FDBNGL   efa_mode_00 , dbcc_displacement
FDBNLE   efa_mode_00 , dbcc_displacement
FDBNLT   efa_mode_00 , dbcc_displacement
FDBNGE   efa_mode_00 , dbcc_displacement
FDBNGT   efa_mode_00 , dbcc_displacement
FDBSNE   efa_mode_00 , dbcc_displacement
FDBST    efa_mode_00 , dbcc_displacement
FDIV     data_modes_1 , fpn
FDIV     fpn , fpn
FETOX    data_modes_1 , fpn
FETOX    fpn , fpn
FETOX    fpn
FETOXM1  data_modes_1 , fpn
FETOXM1  fpn , fpn
FETOXM1  fpn
FGETEXP  data_modes_1 , fpn
FGETEXP  fpn , fpn
FGETEXP  fpn
FGETMAN  data_modes_1 , fpn
FGETMAN  fpn , fpn
FGETMAN  fpn
FINT     data_modes_1 , fpn
FINT     fpn , fpn
FINT     fpn
FINTRZ   data_modes_1 , fpn
FINTRZ   fpn , fpn
FINTRZ   fpn
```

```
FLOG10  data_modes_1 , fpn
FLOG10  fpn , fpn
FLOG10  fpn
FLOG2   data_modes_1 , fpn
FLOG2   fpn , fpn
FLOG2   fpn
FLOGN   data_modes_1 , fpn
FLOGN   fpn , fpn
FLOGN   fpn
FLOGNP1 data_modes_1 , fpn
FLOGNP1 fpn , fpn
FLOGNP1 fpn
FMOD    data_modes_1 , fpn
FMOD    fpn , fpn
FMOVE   data_modes_1 , fpn
FMOVE   fpn , alterable_data_modes
FMOVE   fpn , alterable_data_modes { efa_mode_00 }
FMOVE   fpn , alterable_data_modes { bit_7 }
FMOVECR bit_7 , fpn
FMOVEM  fpr_list , movem_dest_mode
FMOVEM  efa_mode_00 , movem_dest_mode
FMOVEM  movem_src_mode , fpr_list
FMOVEM  movem_src_mode , efa_mode_00
FMOVEM  fpc_list , alterable_modes
FMOVEM  all_modes , fpc_list
FMUL    data_modes_1 , fpn
FMUL    fpn , fpn
FNEG    data_modes_1 , fpn
FNEG    fpn , fpn
FNEG    fpn
FNOP
FREM    data_modes_1 , fpn
FREM    fpn , fpn
FRESTORE movem_src_mode
FSAVE   movem_dest_mode
FSCALE  data_modes_1 , fpn
FSCALE  fpn , fpn
FSF     alterable_data_modes
FSEQ    alterable_data_modes
FSOGT   alterable_data_modes
FSOGE   alterable_data_modes
FSOLT   alterable_data_modes
FSOLE   alterable_data_modes
FSOGL   alterable_data_modes
FSOR    alterable_data_modes
FSON    alterable_data_modes
FSUEQ   alterable_data_modes
FSUGT   alterable_data_modes
FSUGE   alterable_data_modes
```

```
FSULT    alterable_data_modes
FSULE    alterable_data_modes
FSNE     alterable_data_modes
FST      alterable_data_modes
FSSF     alterable_data_modes
FSSEQ    alterable_data_modes
FSGT     alterable_data_modes
FSGE     alterable_data_modes
FSLT     alterable_data_modes
FSLE     alterable_data_modes
FSGL     alterable_data_modes
FSGLE    alterable_data_modes
FSNGLE   alterable_data_modes
FSNGL    alterable_data_modes
FSNLE    alterable_data_modes
FSNLT    alterable_data_modes
FSNGE    alterable_data_modes
FSNGT    alterable_data_modes
FSSNE    alterable_data_modes
FSST     alterable_data_modes
FSGLDIV  data_modes_1 , fpn
FSGLDIV  fpn , fpn
FSGLMUL  data_modes_1 , fpn
FSGLMUL  fpn , fpn
FSIN     data_modes_1 , fpn
FSIN     fpn , fpn
FSIN     fpn
FSINCOS  data_modes_1 , fpn : fpn
FSINCOS  fpn , fpn : fpn
FSINH    data_modes_1 , fpn
FSINH    fpn , fpn
FSINH    fpn
FSQRT    data_modes_1 , fpn
FSQRT    fpn , fpn
FSQRT    fpn
FSUB     data_modes_1 , fpn
FSUB     fpn , fpn
FTAN     data_modes_1 , fpn
FTAN     fpn , fpn
FTAN     fpn
FTANH    data_modes_1 , fpn
FTANH    fpn , fpn
FTANH    fpn
FTENTOX  data_modes_1 , fpn
FTENTOX  fpn , fpn
FTENTOX  fpn
FTRAPF
FTRAPF   # immediate
FTRAPEQ
```

```
FTRAPEQ # immediate
FTRAPOGT
FTRAPOGT    # immediate
FTRAPOGE
FTRAPOGE    # immediate
FTRAPOLT
FTRAPOLT    # immediate
FTRAPOLE
FTRAPOLE    # immediate
FTRAPOGL
FTRAPOGL    # immediate
FTRAPOR
FTRAPOR     # immediate
FTRAPON
FTRAPON     # immediate
FTRAPUEQ
FTRAPUEQ    # immediate
FTRAPUGT
FTRAPUGT    # immediate
FTRAPUGE
FTRAPUGE    # immediate
FTRAPULT
FTRAPULT    # immediate
FTRAPULE
FTRAPULE    # immediate
FTRAPNE
FTRAPNE     # immediate
FTRAPT
FTRAPT      # immediate
FTRAPSF
FTRAPSF     # immediate
FTRAPSEQ
FTRAPSEQ    # immediate
FTRAPGT
FTRAPGT     # immediate
FTRAPGE
FTRAPGE     # immediate
FTRAPLT
FTRAPLT     # immediate
FTRAPLE
FTRAPLE     # immediate
FTRAPGL
FTRAPGL     # immediate
FTRAPGLE
FTRAPGLE    # immediate
FTRAPNGLE
FTRAPNGLE   # immediate
FTRAPNGL
FTRAPNGL    # immediate
```

```
            FTRAPNLE                                                    |
            FTRAPNLE    # immediate                                     |
            FTRAPNLT                                                    |
            FTRAPNLT    # immediate                                     |
            FTRAPNGE                                                    |
            FTRAPNGE    # immediate                                     |
            FTRAPNGT                                                    |
            FTRAPNGT    # immediate                                     |
            FTRAPSNE                                                    |
            FTRAPSNE    # immediate                                     |
            FTRAPST                                                     |
            FTRAPST     # immediate                                     |
            FTST    data_modes_1                                        |
            FTWOTOX data_modes_1 , fpn                                  |
            FTWOTOX fpn , fpn                                           |
            FTWOTOX fpn

all_modes                       --> efa_mode_00 | efa_mode_01 |
                                    efa_mode_02 | efa_mode_03 |
                                    efa_mode_04 | efa_mode_05 |
                                    efa_mode_06 | efa_mode_07 |
                                    efa_mode_08 | efa_mode_09 |
                                    efa_mode_10 | efa_mode_11 |
                                    efa_mode_12 | efa_mode_15 |
                                    efa_mode_16 | efa_mode_17 |
                                    efa_mode_18 | efa_mode_19

alterable_memory_modes          --> efa_mode_02 | efa_mode_03 |
                                    efa_mode_04 | efa_mode_05 |
                                    efa_mode_06 | efa_mode_07 |
                                    efa_mode_08 | efa_mode_09 |
                                    efa_mode_10 | efa_mode_11
```

```
alterable_data_modes          --> efa_mode_00 | efa_mode_02 |
                                  efa_mode_03 | efa_mode_04 |
                                  efa_mode_05 | efa_mode_06 |
                                  efa_mode_07 | efa_mode_08 |
                                  efa_mode_09 | efa_mode_10 |
                                  efa_mode_11

alterable_modes               --> efa_mode_00 | efa_mode_01 |
                                  efa_mode_02 | efa_mode_03 |
                                  efa_mode_04 | efa_mode_05 |
                                  efa_mode_06 | efa_mode_07 |
                                  efa_mode_08 | efa_mode_09 |
                                  efa_mode_10 | efa_mode_11

data_modes_1                  --> efa_mode_00 | efa_mode_02 |
                                  efa_mode_03 | efa_mode_04 |
                                  efa_mode_05 | efa_mode_06 |
                                  efa_mode_07 | efa_mode_08 |
                                  efa_mode_09 | efa_mode_10 |
                                  efa_mode_11 | efa_mode_12 |
                                  efa_mode_15 | efa_mode_16 |
                                  efa_mode_17 | efa_mode_18 |
                                  efa_mode_19

dn_or_alterable_control_modes --> efa_mode_00 bit_field |
                              --> efa_mode_02 bit_field |
                              --> efa_mode_05 bit_field |
                              --> efa_mode_06 bit_field |
                              --> efa_mode_07 bit_field |
                              --> efa_mode_08 bit_field |
                              --> efa_mode_09 bit_field |
                              --> efa_mode_10 bit_field |
                              --> efa_mode_11 bit_field

dn_or_control_modes           --> efa_mode_00 bit_field |
                              --> efa_mode_02 bit_field |
                              --> efa_mode_05 bit_field |
                              --> efa_mode_06 bit_field |
                              --> efa_mode_07 bit_field |
                              --> efa_mode_08 bit_field |
                              --> efa_mode_09 bit_field |
                              --> efa_mode_10 bit_field |
                              --> efa_mode_11 bit_field |
                              --> efa_mode_15 bit_field |
                              --> efa_mode_16 bit_field |
                              --> efa_mode_17 bit_field |
                              --> efa_mode_18 bit_field |
                              --> efa_mode_19 bit_field
```

```
data_modes_2                    --> efa_mode_00 | efa_mode_02 |
                                    efa_mode_03 | efa_mode_04 |
                                    efa_mode_05 | efa_mode_06 |
                                    efa_mode_07 | efa_mode_08 |
                                    efa_mode_09 | efa_mode_10 |
                                    efa_mode_11 | efa_mode_15 |
                                    efa_mode_16 | efa_mode_17 |
                                    efa_mode_18 | efa_mode_19

control_modes                       efa_mode_02 | efa_mode_05 |
                                    efa_mode_06 | efa_mode_07 |
                                    efa_mode_08 | efa_mode_09 |
                                    efa_mode_10 | efa_mode_11 |
                                    efa_mode_15 | efa_mode_16 |
                                    efa_mode_17 | efa_mode_18 |
                                    efa_mode_19

alterable_control_modes         --> efa_mode_02 | efa_mode_05 |
                                    efa_mode_06 | efa_mode_07 |
                                    efa_mode_08 | efa_mode_09 |
                                    efa_mode_10 | efa_mode_11

memory_modes                    --> efa_mode_02 | efa_mode_03 |
                                    efa_mode_04 | efa_mode_05 |
                                    efa_mode_06 | efa_mode_07 |
                                    efa_mode_08 | efa_mode_09 |
                                    efa_mode_10 | efa_mode_11 |
                                    efa_mode_12 | efa_mode_15 |
                                    efa_mode_16 | efa_mode_17 |
                                    efa_mode_18 | efa_mode_19

movem_dest_mode                     efa_mode_02 | efa_mode_04 |
                                    efa_mode_05 | efa_mode_06 |
                                    efa_mode_07 | efa_mode_08 |
                                    efa_mode_09 | efa_mode_10 |
                                    efa_mode_11

movem_src_mode                  --> efa_mode_02 | efa_mode_03 |
                                    efa_mode_05 | efa_mode_06 |
                                    efa_mode_07 | efa_mode_08 |
                                    efa_mode_09 | efa_mode_10 |
                                    efa_mode_11

bit_field       --> { bit_spec : bit_spec }

bit_spec        --> bit_number
                    dn

cas2_efa        --> ( rn ) : ( rn )
```

```
control_register--> SFC | DFS | CACR | USP | VBR | CAAR | MSP | ISP

efa_mode_00     --> dn

efa_mode_01     --> an

efa_mode_02     --> ( an )

efa_mode_03     --> ( an ) +

efa_mode_04     --> - ( an )

efa_mode_05     --> ( disp_16 , an )

efa_mode_06     --> ( disp_08 , an , xn )       |
                    (           an , xn )

efa_mode_07     --> ( bd , an , xn )            |
                    (           xn )            |
                    ( bd      , xn )            |
                    ( bd , an     )

efa_mode_08     --> ( [ bd , an , xn ] , od )   |
                    ( [      an , xn ] , od )   |
                    ( [           xn ] , od )   |
                    ( [      an      ] , od )   |
                    ( [           xn ]      )   |
                    ( [      an      ]      )   |
                    ( [ bd      , xn ] , od )   |
                    ( [ bd           ] , od )   |
                    ( [ bd      , xn ]      )   |
                    ( [ bd           ]      )   |
                    ( [ bd , an      ] , od )   |
                    ( [ bd , an      ]      )   |
                    ( [ bd , an , xn ]      )   |
                    ( [      an , xn ]      )

efa_mode_09     --> ( [ bd , an ] , od , xn )   |
                    ( [ bd      ] , xn      )   |
                    ( [ bd      ] , od , xn )   |
                    ( [ bd , an ] , xn      )   |
                    ( [      an ] , xn      )   |
                    ( [      an ] , od , xn )

efa_mode_10     --> ( immediate )

efa_mode_11     --> ( immediate )
```

```
efa_mode_12     --> # immediate

efa_mode_15     --> ( disp_16 , pc )

efa_mode_16     --> ( disp_08 , pc , xn )       |
                    (           pc , xn )

efa_mode_17     --> ( bd , pc , xn )            |
                    ( bd , pc      )

efa_mode_18     --> ( [ bd , pc , xn ] , od )   |
                    ( [ bd , pc , xn ]      )   |
                    ( [ bd , pc      ] , od )   |
                    ( [ bd , pc      ]      )   |
                    ( [      pc , xn ] , od )   |
                    ( [      pc , xn ]      )

efa_mode_19     --> ( [ bd , pc ] , od , xn )   |
                    ( [ bd , pc ] , xn      )

pc              --> PC | ZPC

bd              --> immediate_range_0_to_FFFFFFFF

od              --> immediate_range_0_to_FFFFFFFF

xn              --> index | index * scale

index           --> D0.W | D1.W | D2.W | D3.W |
                    D4.W | D5.W | D6.W | D7.W |
                    D0.L | D1.L | D2.L | D3.L |
                    D4.L | D5.L | D6.L | D7.L |
                    A0.W | A1.W | A2.W | A3.W |
                    A4.W | A5.W | A6.W | A7.W |
                    A0.L | A1.L | A2.L | A3.L |
                    A4.L | A5.L | A6.L | A7.L |
                    SP.W | SP.L |
                    dn | an
scale           --> expression

register_list   --> register_range {/ register_range}

register_range  --> rn | rn - rn

rn              --> an | dn

register_pair   --> dn : dn

an              --> A0 | A1 | A2 | A3 | A4 | A5 | A6 | A7 | SP
```

```
dn                  --> D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7

sr_or_ccr           --> SR | CCR

usp                 --> USP

dbcc_displacement                          --> expression

branch_displacement                        --> expression

bit_number                                 --> expression

immediate                                  --> expression

immediate_bit_16                           --> expression

immediate_range_0_15                       --> expression

immediate_range_0_255                      --> expression

immediate_range_0_7                        --> expression

immediate_range_1_to_8                     --> expression

immediate_range_m128_to_127                --> expression

immediate_range_m32768_to_32767            --> expression

immediate_range_0_to_FFFFFFFF              --> expression

disp_08                                    --> expression

disp_16                                    --> expression

bit_3                                      --> expression

bit_4                                      --> expression

bit_7                                      --> expression

mmufc               --> bit_4 | DN | SFC | DFC

mmu_reg             --> CRP   | DRP   | TC    | AC    |
                        PSR   | PCSR  | CAL   | VAL   | SCC |
                        BAC0  | BAC1  | BAC2  | BAC3  |
                        BAC4  | BAC5  | BAC6  | BAC7  |
                        BAD0  | BAD1  | BAD2  | BAD3  |
                        BAD4  | BAD5  | BAD6  | BAD7
```

```
fpn             --> FP0 | FP1 | FP2 | FP3 | FP4 | FP5 | FP6 | FP7

fpc_list        --> fpc_register {/ fpc_register}

fpc_register    --> FPCR | FPSR | FPIAR

fpr_list        --> fpc_range {/ fpc_range}

fpc_range       --> fpn | fpn - fpn
```

# Appendix IV: Compiler Runtime Interface

In this appendix is a representative Ada specification for the runtime interface of the compiler. This is presented for information only; it does not match the actual implementation and user code should not depend on this specification.

```
package Runtime_Interface is

    type Address is private;

    type Tcb is private;

    type Tcb_Array is array (Positive range <>) of Tcb;

    type Layer is private;

    type Exception_Id is private;

    package Attributes is

        type String_Dope_Vector is private;

        type Enumeration_Image_Table is private;
        type Enumeration_Representation_Table (Count : Positive) is private;

        procedure Enum_Image (Table : in Enumeration_Image_Table;
                              Value : in Integer;
                              Str : out String;
                              Dope : out String_Dope_Vector);
        pragma Suppress (Elaboration_Check, On => Enum_Image);
        pragma Interface (Asm, Enum_Image);
        pragma Import_Procedure
           (Internal => Enum_Image,
            External => "__ENUM_IMAGE",
            Mechanism => (Reference, Value, Reference, Reference));

        function Enum_Pos (Table : in Enumeration_Representation_Table;
                           Value : in Integer) return Integer;
        pragma Suppress (Elaboration_Check, On => Enum_Pos);
```

```
pragma Interface (Asm, Enum_Pos);
pragma Import_Function (Internal => Enum_Pos,
                        External => "__ENUM_POS",
                        Mechanism => (Reference, Value));

function Enum_Pred (Table : in Enumeration_Representation_Table;
                    Value : in Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Enum_Pred);
pragma Interface (Asm, Enum_Pred);
pragma Import_Function (Internal => Enum_Pred,
                        External => "__ENUM_PRED",
                        Mechanism => (Reference, Value));

function Enum_Succ (Table : in Enumeration_Representation_Table;
                    Value : in Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Enum_Succ);
pragma Interface (Asm, Enum_Succ);
pragma Import_Function (Internal => Enum_Succ,
                        External => "__ENUM_SUCC",
                        Mechanism => (Reference, Value));

function Enum_Value (Table : in Enumeration_Image_Table;
                     Length : in Integer;
                     Str : in String) return Integer;
pragma Suppress (Elaboration_Check, On => Enum_Value);
pragma Interface (Asm, Enum_Value);
pragma Import_Function (Internal => Enum_Value,
                        External => "__ENUM_VALUE",
                        Mechanism => (Reference, Value, Reference));

function Enum_Width (Table : in Enumeration_Image_Table;
                     Low_Bound : in Integer;
                     High_Bound : in Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Enum_Width);
pragma Interface (Asm, Enum_Width);
pragma Import_Function (Internal => Enum_Width,
                        External => "__ENUM_WIDTH",
                        Mechanism => (Reference, Value, Value));

procedure Int_Image (Value : in Integer;
                     Str : out String;
                     Dope : out String_Dope_Vector);
pragma Suppress (Elaboration_Check, On => Int_Image);
pragma Interface (Asm, Int_Image);
pragma Import_Procedure (Internal => Int_Image,
                         External => "__INT_IMAGE",
                         Mechanism => (Value, Reference, Reference));
```

```
        function Int_Value (Length : in Integer;
                            Str : in String) return Integer;
        pragma Suppress (Elaboration_Check, On => Int_Value);
        pragma Interface (Asm, Int_Value);
        pragma Import_Function (Internal => Int_Value,
                                External => "__INT_VALUE",
                                Mechanism => (Value, Reference));

        function Int_Width (Low_Bound : in Integer;
                            High_Bound : in Integer) return Integer;
        pragma Suppress (Elaboration_Check, On => Int_Width);
        pragma Interface (Asm, Int_Width);
        pragma Import_Function (Internal => Int_Width,
                                External => "__INT_WIDTH",
                                Mechanism => (Value, Value));

--      Boolean_Image : constant Enumeration_Image_Table;
--      pragma Import_Object (Internal => Boolean_Image,
--                            External => "__BOOLEAN_IMAGE");
--      Character_Image : constant Enumeration_Image_Table;
--      pragma Import_Object (Internal => Character_Image,
--                            External => "__CHARACTER_IMAGE");

    private

        type Enumeration_Image_Table is new Integer;
        --
        --  The actual contents of an Enumeration_Image_Table cannot be
        --  described in Ada.  The table consists of a 16-bit element count
        --  that is followed by a group of 16-bit displacements.  The
        --  image strings immediately follow this table of displacements.
        --  The image string for a given enumeration is found by taking
        --  the 'Pos of the enumeration and indexing into the displacements.
        --  The value thus found is added to the address of the base of
        --  the table to form the address of the first character in the
        --  string that is its image.  The length of the image for a given
        --  enumeration is found by subtracting its displacement from the
        --  displacement of the following item.
        --
        --  On a machine with byte addressing, a table would look like this:
        --
        --      type T is (A, Bb, Ccc, Dddd);
        --
        --      enum_table:
        --              .dc.w   4               ; 4 elements
        --              .dc.w   12              ; Offset from table to A
        --              .dc.w   13              ; Offset from table to BB
        --              .dc.w   15              ; Offset from table to CCC
```

```
--              .dc.w   18              ; Offset from table to DDDD
--              .dc.w   22              ; For finding length of 'Last
--              .ascii  "ABBCCCDDDD"    ; Always uppercase

    type Enumeration_Representations is
       array (Positive range <>) of Integer;
    type Enumeration_Representation_Table (Count : Positive) is
        record
            Representations : Enumeration_Representations (1 .. Count);
        end record;

    type String_Dope_Vector is
        record
            Length : Integer;
            First : Integer;
            Last : Integer;
        end record;

end Attributes;

package Exceptions is

    procedure Raise_Exception (Id : in Exception_Id);
    pragma Suppress (Elaboration_Check, On => Raise_Exception);
    pragma Interface (Asm, Raise_Exception);
    pragma Import_Procedure (Internal => Raise_Exception,
                             External => "__RAISE_EXCEPTION",
                             Mechanism => (Value));

    procedure Raise_Constraint_Error;
    pragma Suppress (Elaboration_Check, On => Raise_Constraint_Error);
    pragma Interface (Asm, Raise_Constraint_Error);
    pragma Import_Procedure (Internal => Raise_Constraint_Error,
                             External => "__RAISE_CONSTRAINT_ERROR");

    procedure Propagate_Exception;
    pragma Suppress (Elaboration_Check, On => Propagate_Exception);
    pragma Interface (Asm, Propagate_Exception);
    pragma Import_Procedure (Internal => Propagate_Exception,
                             External => "__PROPAGATE_EXCEPTION");

    procedure Stack_Check (Storage_Units : Natural);
    pragma Suppress (Elaboration_Check, On => Stack_Check);
    pragma Interface (Asm, Stack_Check);
    pragma Import_Procedure (Internal => Stack_Check,
                             External => "__STACK_CHECK",
                             Mechanism => (Value));

end Exceptions;
```

```
package Storage_Management is

    type Collection is private;

    function Allocate_Collection
                (Storage_Units : in Integer;
                 Extensible : in Boolean) return Collection;
    pragma Suppress (Elaboration_Check, On => Allocate_Collection);
    pragma Interface (Asm, Allocate_Collection);
    pragma Import_Function (Internal => Allocate_Collection,
                            External => "__ALLOCATE_COLLECTION",
                            Mechanism => (Value, Value));

    function Allocate_Fixed_Cell
                (Storage_Units : in Integer;
                 From_Collection : in Collection) return Address;
    pragma Suppress (Elaboration_Check, On => Allocate_Fixed_Cell);
    pragma Interface (Asm, Allocate_Fixed_Cell);
    pragma Import_Function (Internal => Allocate_Fixed_Cell,
                            External => "__ALLOCATE_FIXED_CELL",
                            Mechanism => (Value, Reference));

    procedure Deallocate_Collection (The_Collection : in out Collection);
    pragma Suppress (Elaboration_Check, On => Deallocate_Collection);
    pragma Interface (Asm, Deallocate_Collection);
    pragma Import_Procedure (Internal => Deallocate_Collection,
                             External => "__DEALLOCATE_COLLECTION",
                             Mechanism => (Reference));

    procedure Deallocate_Fixed_Cell (Storage_Units : in Integer;
                                     To_Collection : in Collection;
                                     Cell : in Address);
    pragma Suppress (Elaboration_Check, On => Deallocate_Fixed_Cell);
    pragma Interface (Asm, Deallocate_Fixed_Cell);
    pragma Import_Procedure (Internal => Deallocate_Fixed_Cell,
                             External => "__DEALLOCATE_FIXED_CELL",
                             Mechanism => (Value, Value, Reference));

    function Collection_Size (Of_Collection : in Collection) return
                                                              Integer;
    pragma Suppress (Elaboration_Check, On => Collection_Size);
    pragma Interface (Asm, Collection_Size);
    pragma Import_Function (Internal => Collection_Size,
                            External => "__COLLECTION_SIZE",
                            Mechanism => (Value));
```

```
    private

        type Collection_Information is
            record
                null;
            end record;

        type Collection is access Collection_Information;
        for Collection'Storage_Size use 0;

    end Storage_Management;

    package Tasking is

--      Top_Layer : constant Layer;
--      pragma Import_Object (Internal => Top_Layer,
                              External => "__TOP_LAYER");

        package Task_Management is

            procedure Initialize_Master (Master_Layer : in Layer);
            pragma Suppress (Elaboration_Check, On => Initialize_Master);
            pragma Interface (Asm, Initialize_Master);
            pragma Import_Procedure (Internal => Initialize_Master,
                                     External => "__INITIALIZE_MASTER",
                                     Mechanism => (Reference));

            function Create_Task (Activation_Group : in Tcb;
                                  Master_Layer : in Layer;
                                  Start_Pc : in Address;
                                  Entry_Count : in Integer;
                                  Stack_Size : in Integer;
                                  Priority : in Integer;
                                  Frame_Pointer : in Address) return Tcb;
            pragma Suppress (Elaboration_Check, On => Create_Task);
            pragma Interface (Asm, Create_Task);
            pragma Import_Function
               (Internal => Create_Task,
                External => "__CREATE_TASK",
                Mechanism => (Reference, Reference, Reference,
                              Value, Value, Value, Value));

            procedure Activate_Offspring
                        (Activation_Group : in out Tcb;
                         Perform_Elaboration_Check : in Boolean);
            pragma Suppress (Elaboration_Check, On => Activate_Offspring);
            pragma Interface (Asm, Activate_Offspring);
```

```
        pragma Import_Procedure (Internal => Activate_Offspring,
                                 External => "__ACTIVATE_OFFSPRING",
                                 Mechanism => (Reference, Value));

        procedure Notify_Parent (Task_Error : in Boolean);
        pragma Suppress (Elaboration_Check, On => Notify_Parent);
        pragma Interface (Asm, Notify_Parent);
        pragma Import_Procedure (Internal => Notify_Parent,
                                 External => "__NOTIFY_PARENT",
                                 Mechanism => (Value));

        procedure Task_End;
        pragma Suppress (Elaboration_Check, On => Task_End);
        pragma Interface (Asm, Task_End);
        pragma Import_Procedure (Internal => Task_End,
                                 External => "__TASK_END");

        procedure Await_Dependents;
        pragma Suppress (Elaboration_Check, On => Await_Dependents);
        pragma Interface (Asm, Await_Dependents);
        pragma Import_Procedure (Internal => Await_Dependents,
                                 External => "__AWAIT_DEPENDENTS");

        procedure Task_Completion;
        pragma Suppress (Elaboration_Check, On => Task_Completion);
        pragma Interface (Asm, Task_Completion);
        pragma Import_Procedure (Internal => Task_Completion,
                                 External => "__TASK_COMPLETION");

        procedure Terminate_Allocated_Offspring
                     (Activation_Group : in out Tcb);
        pragma Suppress (Elaboration_Check,
                         On => Terminate_Allocated_Offspring);
        pragma Interface (Asm, Terminate_Allocated_Offspring);
        pragma Import_Procedure
           (Internal => Terminate_Allocated_Offspring,
            External => "__TERMINATE_ALLOCATED_OFFSPRING",
            Mechanism => (Reference));

        procedure Terminate_Dependent_Offspring;
        pragma Suppress (Elaboration_Check,
                         On => Terminate_Dependent_Offspring);
        pragma Interface (Asm, Terminate_Dependent_Offspring);
        pragma Import_Procedure
           (Internal => Terminate_Dependent_Offspring,
            External => "__TERMINATE_DEPENDENT_OFFSPRING");

    end Task_Management;
```

```
package Rendezvous is

    package Entry_Calls is

        procedure Entry_Call (Tsk : in Tcb;
                              Entry_Number : in Positive;
                              Parameters : in Address);
        pragma Suppress (Elaboration_Check, On => Entry_Call);
        pragma Interface (Asm, Entry_Call);
        pragma Import_Procedure (Internal => Entry_Call,
                                 External => "__ENTRY_CALL",
                                 Mechanism => (Value, Value, Value));

        function Conditional_Entry_Call
                    (Tsk : in Tcb;
                     Entry_Number : in Positive;
                     Parameters : in Address) return Boolean;
        pragma Suppress (Elaboration_Check,
                         On => Conditional_Entry_Call);
        pragma Interface (Asm, Conditional_Entry_Call);
        pragma Import_Function (Internal => Conditional_Entry_Call,
                                External =>
                                        "__CONDITIONAL_ENTRY_CALL",
                                Mechanism => (Value, Value, Value));

        function Timed_Entry_Call
                    (Tsk : in Tcb;
                     Entry_Number : in Positive;
                     Timeout : in Duration;
                     Parameters : in Address) return Boolean;
        pragma Suppress (Elaboration_Check, On => Timed_Entry_Call);
        pragma Interface (Asm, Timed_Entry_Call);
        pragma Import_Function
           (Internal => Timed_Entry_Call,
            External => "__TIMED_ENTRY_CALL",
            Mechanism => (Value, Value, Value, Value));

    end Entry_Calls;

    package Accepts is

        procedure Begin_Accept (Entry_Number : in Positive;
                                Parameters : in out Address);
        pragma Suppress (Elaboration_Check, On => Begin_Accept);
        pragma Interface (Asm, Begin_Accept);
        pragma Import_Procedure (Internal => Begin_Accept,
                                 External => "__BEGIN_ACCEPT",
                                 Mechanism => (Value, Value));
```

```
        procedure End_Accept (Propagate_Exception : in Boolean);
        pragma Suppress (Elaboration_Check, On => End_Accept);
        pragma Interface (Asm, End_Accept);
        pragma Import_Procedure (Internal => End_Accept,
                                 External => "__END_ACCEPT",
                                 Mechanism => (Value));

        procedure Quick_Accept (Entry_Number : in Positive);
        pragma Suppress (Elaboration_Check, On => Quick_Accept);
        pragma Interface (Asm, Quick_Accept);
        pragma Import_Procedure (Internal => Quick_Accept,
                                 External => "__QUICK_ACCEPT",
                                 Mechanism => (Value));

    end Accepts;

    package Selective_Waits is

        type Select_Forms is (Terminate_Alternative,
                              Delay_Alternative,
                              Else_Part);
        for Select_Forms use (Terminate_Alternative => -1,
                              Delay_Alternative => 0,
                              Else_Part => 1);

        type Entry_Number is range 0 .. 32767;

        type Branch_Number is range 0 .. 32767;

        type Arm_Info is
            record
                Is_Simple_Accept : Boolean;
                For_Entry : Entry_Number;
            end record;

        for Arm_Info use
            record
                Is_Simple_Accept at 0 range 0 .. 0;
                For_Entry at 0 range 1 .. 15;
            end record;

        type Entry_Map is array (Branch_Number range <>) of Arm_Info;

        type Select_Information (Arms : Branch_Number) is
            record
                Form : Select_Forms;
                Delay_Arm : Branch_Number;
                Delay_Time : Duration;
                Map : Entry_Map (1 .. Arms);
```

```
            end record;

        for Select_Information use
            record
                Form at 0 range 0 .. 15;
                Arms at 0 range 16 .. 31;
                Delay_Time at 0 range 32 .. 63;
                Delay_Arm at 0 range 64 .. 79;
            end record;

        function Select_Rendezvous
                    (Info : Select_Information;
                     Round_Robin : Positive;
                     Params : Address) return Branch_Number;
        pragma Suppress (Elaboration_Check, On => Select_Rendezvous);
        pragma Interface (Asm, Select_Rendezvous);
        pragma Import_Function
           (Internal => Select_Rendezvous,
            External => "__SELECT_RENDEZVOUS",
            Mechanism => (Reference, Reference, Value));

    end Selective_Waits;

end Rendezvous;

package Miscellaneous is

    procedure Abort_Multiple_Tasks (Count : in Positive;
                                    Tasks : in Tcb_Array);
    pragma Suppress (Elaboration_Check, On => Abort_Multiple_Tasks);
    pragma Interface (Asm, Abort_Multiple_Tasks);
    pragma Import_Procedure (Internal => Abort_Multiple_Tasks,
                             External => "__ABORT_MULTIPLE_TASKS",
                             Mechanism => (Value, Value));

    function Check_Return_Task
                (Tsk : in Tcb; Frame : in Address) return Boolean;
    pragma Suppress (Elaboration_Check, On => Check_Return_Task);
    pragma Interface (Asm, Check_Return_Task);
    pragma Import_Function (Internal => Check_Return_Task,
                            External => "__CHECK_RETURN_TASK",
                            Mechanism => (Value, Value));

    procedure Delay_Statement (Timeout : in Duration);
    pragma Suppress (Elaboration_Check, On => Delay_Statement);
    pragma Interface (Asm, Delay_Statement);
    pragma Import_Procedure (Internal => Delay_Statement,
                             External => "__DELAY_STATEMENT",
                             Mechanism => (Value));
```

```
        function Entry_Count (Entry_Number : in Positive) return Integer;
        pragma Suppress (Elaboration_Check, On => Entry_Count);
        pragma Interface (Asm, Entry_Count);
        pragma Import_Function (Internal => Entry_Count,
                                External => "__ENTRY_COUNT",
                                Mechanism => (Value));

        function Task_Callable (Tsk : in Tcb) return Boolean;
        pragma Suppress (Elaboration_Check, On => Task_Callable);
        pragma Interface (Asm, Task_Callable);
        pragma Import_Function (Internal => Task_Callable,
                                External => "__TASK_CALLABLE",
                                Mechanism => (Value));

        function Task_Stack_Size (Tsk : in Tcb) return Integer;
        pragma Suppress (Elaboration_Check, On => Task_Stack_Size);
        pragma Interface (Asm, Task_Stack_Size);
        pragma Import_Function (Internal => Task_Stack_Size,
                                External => "__TASK_STACK_SIZE",
                                Mechanism => (Value));

        function Task_Terminated (Tsk : in Tcb) return Boolean;
        pragma Suppress (Elaboration_Check, On => Task_Terminated);
        pragma Interface (Asm, Task_Terminated);
        pragma Import_Function (Internal => Task_Terminated,
                                External => "__TASK_TERMINATED",
                                Mechanism => (Value));
    end Miscellaneous;

end Tasking;

package Miscellaneous is

    procedure Start_Sequential (Return_Pc : in Address);
    pragma Suppress (Elaboration_Check, On => Start_Sequential);
    pragma Interface (Asm, Start_Sequential);
    pragma Import_Procedure (Internal => Start_Sequential,
                             External => "__START_SEQUENTIAL",
                             Mechanism => (Value));

    procedure Middle_Sequential;
    pragma Suppress (Elaboration_Check, On => Middle_Sequential);
    pragma Interface (Asm, Middle_Sequential);
    pragma Import_Procedure (Internal => Middle_Sequential,
                             External => "__MIDDLE_SEQUENTIAL");

    procedure Finish_Sequential;
```

```
        pragma Suppress (Elaboration_Check, On => Finish_Sequential);
        pragma Interface (Asm, Finish_Sequential);
        pragma Import_Procedure (Internal => Finish_Sequential,
                                 External => "__FINISH_SEQUENTIAL");

        procedure Start_Tasking (Return_Pc : in Address);
        pragma Suppress (Elaboration_Check, On => Start_Tasking);
        pragma Interface (Asm, Start_Tasking);
        pragma Import_Procedure (Internal => Start_Tasking,
                                 External => "__START_TASKING",
                                 Mechanism => (Value));

        procedure Middle_Tasking;
        pragma Suppress (Elaboration_Check, On => Middle_Tasking);
        pragma Interface (Asm, Middle_Tasking);
        pragma Import_Procedure (Internal => Middle_Tasking,
                                 External => "__MIDDLE_TASKING");

        procedure Finish_Tasking;
        pragma Suppress (Elaboration_Check, On => Finish_Tasking);
        pragma Interface (Asm, Finish_Tasking);
        pragma Import_Procedure (Internal => Finish_Tasking,
                                 External => "__FINISH_TASKING");

        type Debug_Info_Table is
            record
                null;
            end record;

        Debug_Info_Sequential : Debug_Info_Table;
        pragma Import_Object (Internal => Debug_Info_Sequential,
                              External => "__DEBUG_INFO_SEQUENTIAL");

        Debug_Info_Tasking : Debug_Info_Table;
        pragma Import_Object (Internal => Debug_Info_Tasking,
                              External => "__DEBUG_INFO_TASKING");

    end Miscellaneous;

    package Block_Compare is

        function Compare_8_Bit_Signed (L : Address;
                                       R : Address;
                                       L_Length : Integer;
                                       R_Length : Integer) return Integer;
        pragma Suppress (Elaboration_Check, On => Compare_8_Bit_Signed);
        pragma Interface (Asm, Compare_8_Bit_Signed);
        pragma Import_Function (Internal => Compare_8_Bit_Signed,
                                External => "__COMPARE_8_BIT_SIGNED",
```

```
                        Mechanism => (Value, Value, Value, Value));

function Compare_16_Bit_Signed (L : Address;
                                R : Address;
                                L_Length : Integer;
                                R_Length : Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Compare_16_Bit_Signed);
pragma Interface (Asm, Compare_16_Bit_Signed);
pragma Import_Function (Internal => Compare_16_Bit_Signed,
                        External => "__COMPARE_16_BIT_SIGNED",
                        Mechanism => (Value, Value, Value, Value));

function Compare_32_Bit_Signed (L : Address;
                                R : Address;
                                L_Length : Integer;
                                R_Length : Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Compare_32_Bit_Signed);
pragma Interface (Asm, Compare_32_Bit_Signed);
pragma Import_Function (Internal => Compare_32_Bit_Signed,
                        External => "__COMPARE_32_BIT_SIGNED",
                        Mechanism => (Value, Value, Value, Value));

function Compare_8_Bit_Unsigned (L : Address;
                                 R : Address;
                                 L_Length : Integer;
                                 R_Length : Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Compare_8_Bit_Unsigned);
pragma Interface (Asm, Compare_8_Bit_Unsigned);
pragma Import_Function (Internal => Compare_8_Bit_Unsigned,
                        External => "__COMPARE_8_BIT_UNSIGNED",
                        Mechanism => (Value, Value, Value, Value));

function Compare_16_Bit_Unsigned (L : Address;
                                  R : Address;
                                  L_Length : Integer;
                                  R_Length : Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Compare_16_Bit_Unsigned);
pragma Interface (Asm, Compare_16_Bit_Unsigned);
pragma Import_Function (Internal => Compare_16_Bit_Unsigned,
                        External => "__COMPARE_16_BIT_UNSIGNED",
                        Mechanism => (Value, Value, Value, Value));

function Compare_32_Bit_Unsigned (L : Address;
                                  R : Address;
                                  L_Length : Integer;
                                  R_Length : Integer) return Integer;
pragma Suppress (Elaboration_Check, On => Compare_32_Bit_Unsigned);
pragma Interface (Asm, Compare_32_Bit_Unsigned);
pragma Import_Function (Internal => Compare_32_Bit_Unsigned,
```

```
                                External => "__COMPARE_32_BIT_UNSIGNED",
                                Mechanism => (Value, Value, Value, Value));

    end Block_Compare;

private

    type Addressable_Object;
    type Address is access Addressable_Object;
    for Address'Storage_Size use 0;

    type Task_Control_Block (Entries : Natural);

    type Tcb is access Task_Control_Block;
    for Tcb'Storage_Size use 0;

    type Layer_Information;

    type Layer is access Layer_Information;
    for Layer'Storage_Size use 0;

    type Exception_Id is new Address;

end Runtime_Interface;
```

# Appendix V: Appendix F to the LRM for the Mc68020_Os2000 Target

The *Reference Manual for the Ada Programming Language* (LRM) specifies that certain features of the language are implementation-dependent. It requires that these implementation dependencies be defined in an appendix called Appendix F. This is Appendix F for the Mc68020_Os2000 target, compiler version 4. It contains materials on the following topics listed for inclusion by the LRM on page F-1:

- Implementation-dependent pragmas
- Implementation-dependent attributes
- Package System
- Representation clauses
- Implementation-dependent components
- Interpretation of expressions that appear in address clauses
- Unchecked conversion
- Implementation-dependent characteristics of I/O packages

These topics appear in section and subsection titles of this appendix. The appendix contains other topics mentioned in the LRM as being implementation-dependent. For these, a reference to the LRM is given in the section or subsection title. In addition, this appendix contains sections on predefined pragmas and size of objects. This material is included here because of implementation dependencies and close relationship to LRM-mandated topics.

## IMPLEMENTATION-DEPENDENT PRAGMAS

The MC68020/OS-2000 cross-compiler supports pragmas for application software development in addition to those listed in Annex B of the LRM.

### Pragma Main

A parameterless library-unit procedure without subunits can be designated as a main program by including a pragma Main at the end of the unit specification or body. This pragma causes the linker to run and create an executable program when the body of this subprogram is coded. Before a unit having a pragma Main can be coded, all units in the *with* closure of the unit must be coded.

The pragma Main has three arguments:

- **Target**: A string specifying the target key. If this argument appears and it does not match the current target key, the pragma Main is ignored. If the Target parameter matches the current target key or does not appear, pragma Main is honored. A single source copy of a main program can be used for different targets by putting in multiple Main pragmas with different target parameters and different stack sizes and/or different heap sizes.
- **Stack_Size**: A static integer expression specifying the size in bytes of the main task stack. If not specified, the default value is 4096 (4K) bytes.
- **Heap_Size**: A static integer expression specifying the size in bytes of the heap. If not specified, the default value is 64K bytes.

The complete syntax for this pragma is:

```
pragma_main   ::=  PRAGMA MAIN
                   [ ( main_option { , main_option } ) ] ;

main_option   ::=  TARGET     => simple_name |
                   STACK_SIZE => static_integer_expression |
                   HEAP_SIZE  => static_integer_expression
```

The pragma Main must appear immediately after the declaration or body of a parameterless library-unit procedure without subunits.

### Using the Target Parameter

Using the Target parameter forces the pragma to be ignored for all targets but the one specified. This enables joined views of a procedure to have different effects according to the target. One use is to avoid the effects of declaring a pragma Main when the target is Rational:

```
pragma Main (Target => Mc68020_Os2000);
```

Another use is to specify different stack or heap sizes for different targets. For example:

```
procedure Show_Pragma_Main is
begin
    Do_Something;
end Show_Pragma_Main;
pragma Main (Target => Mc68020_Os2000,        Heap_Size => <10*1024>);
pragma Main (Target => <another target key>, Heap_Size => <20*1024>);
```

The procedure Show_Pragma_Main will be a main program in both an Mc68020_Os2000 view and a view for the other target. The heap sizes for the two targets will be as specified by the different pragma Mains.

Multiple pragma Mains may be placed in the specification, the body, or both. If more than one pragma Main is specified with the same target parameter, only one of the pragmas will have any effect. The first pragma Main in the specification (if there is one) will be chosen; otherwise, the first one in the body will be chosen.

## Pragmas Import_Procedure and Import_Function

A subprogram written in another language (typically, assembly language) can be called from an Ada program if it is declared with a pragma Interface. The rules for placement of pragma Interface are given in Section 13.9 of the LRM. Every interfaced subprogram must have an importing pragma that is recognized by the MC68020/OS-2000 cross-compiler, either Import-_Procedure or Import_Function. These pragmas are used to declare the external name of the subprogram and the parameter-passing mechanism for the subprogram call. If an interfaced subprogram does not have an importing pragma, or if the importing pragma is incorrect, pragma Interface is ignored.

Importing pragmas can be applied only to nongeneric procedures and functions.

The pragmas Import_Procedure and Import_Function are used for importing subprograms. Import_Procedure is used to call a non-Ada procedure; Import_Function, a non-Ada function.

Each import pragma must be preceded by a pragma Interface; otherwise, the placement rules for these pragmas are identical to those of the pragma Interface.

The importing pragmas have the form:

```
importing_pragma  ::=  PRAGMA importing_type
                       ( [ INTERNAL => ] internal_name
                         [ , [ EXTERNAL => ] external_name ]
                         [ [ , [ PARAMETER_TYPES => ]
                                                parameter_types ]
                           [ , [ RESULT_TYPE => ] type_mark ] ]
                           [ , NICKNAME => string_literal ]
                         [ , [ MECHANISM => ] mechanisms ] ) ;

importing_type    ::=  IMPORT_PROCEDURE | IMPORT_FUNCTION |
                       IMPORT_VALUED_PROCEDURE
internal_name     ::=  identifier |
                       string_literal   -- An operator designator

external_name     ::=  string_literal

parameter_types   ::=  NULL | ( type_mark { , type_mark } )

mechanisms        ::=  mechanism_name |
                       ( mechanism_name { , mechanism_name } )

mechanism_name    ::=  VALUE | REFERENCE
```

The internal name is the Ada name of the subprogram being interfaced. If more than one subprogram is in the declarative region preceding the importing pragma, the correct subprogram must be identified by either using the argument types (and result type, if a function) or specifying the nickname. See pragma Nickname below.

The purpose of the Parameter_Types argument is to distinguish among two or more overloaded subprograms having the same internal name. The value of the Parameter_Types argument is a list of type or subtype names separated by commas and enclosed in parentheses. Each name corresponds positionally to a formal parameter in the subprogram's declaration. If the subprogram has no parameters, the list consists of the single word *null*. The Result_Type argument serves the same purpose for functions; its value is the type returned by the function.

The external designator, specified with the External parameter, is a character string that is an identifier suitable for the MC68020 assembler. If the external designator is not specified, the internal name is used.

The Mechanism argument is required if the subprogram has any parameters. The argument specifies, in a parenthesized list, the passing mechanism for each parameter to be passed. There must be a mechanism specified for each parameter listed in Parameter_Types and they must correspond positionally. The types of mechanism are as follows:

- Value: Specifies that the parameter is passed on the stack by value.
- Reference: Specifies that the address of the parameter is passed on the stack.

For functions, it is not possible to specify the passing mechanism of the function result; the standard Ada mechanism for the given type of the function result must be used by the interfaced subprogram. If there are parameters, and they all use the same passing mechanism, an alternate form for the Mechanism argument can be used: instead of a parenthesized list with an element for each parameter, the single mechanism name (not parenthesized) can be used.

Examples:

```
procedure Locate (Source: in String;
                  Target: in String;
                  Index:  out Natural);

pragma Interface (Assembler, Locate);
pragma Import_Procedure
           (Internal        => Locate,
            External        => "STR$LOCATE",
            Parameter_Types => (String, String, Natural),
            Mechanism       => (Reference, Reference, Value));


function Pwr (I: Integer; N: Integer) return Float;
function Pwr (F: Float; N: Integer) return Float;

pragma Interface (Assembler, Pwr);

pragma Import_Function
           (Internal        => Pwr,
            Parameter_Types => (Integer, Integer),
            Result_Type     => Float,
            Mechanism       => Value,
            External        => "MATH$PWR_OF_INTEGER");
```

```
pragma Import_Function
           (Internal        => Pwr,
            Parameter_Types => (Float, Integer),
            Result_Type     => Float,
            Mechanism       => Value,
            External        => "MATH$PWR_OF_FLOAT");
```

## Pragmas Export_Procedure and Export_Function

A subprogram written in Ada can be made accessible to code written in another language by using an exporting pragma defined by the MC68020/OS-2000 cross-compiler. The effect of such a pragma is to give the subprogram a global symbolic name that the linker can use when resolving references between object modules.

Exporting pragmas can be applied only to nongeneric procedures and functions.

Exporting a subprogram does not export the mechanism used by the compiler to perform elaboaration checks; calls from other languages to an exported subprogram whose body is not yet elaboarated may have unpredictable results when the subprogram body references objects that are not yet elaborated. Elaboration checks within the Ada program are not affected by the exporting pragma.

An exporting pragma can be given only for a subprogram that is a library unit or that is declared in the specification of a library package. An exporting pragma can be placed after a subprogram body only if the subprogram does not have a separate specification. Thus, an exporting pragma cannot be applied to the body of a library subprogram that has a separate specification.

These pragmas have arguments similar to those of the importing pragmas, except that it is not possible to specify the parameter-passing mechanism. The standard Ada parameter-passing mechanisms are chosen. For descriptions of the pragma's arguments (Internal, External, Parameter_Types, Result_Type, and Nickname), see the preceding section on the importing pragmas.

The full syntax of the pragmas for exporting subprograms is:

```
exporting_pragma  ::=  PRAGMA exporting_type
                       ( [ INTERNAL => ] internal_name
                         [ , [ EXTERNAL => ] external_name ]
                         [ [ , [ PARAMETER_TYPES => ] parameter_types ]
                           [ , [ RESULT_TYPE => ] type_mark ] |
                           [ , NICKNAME => string_literal ] ] ) ;
exporting_type    ::=  EXPORT_PROCEDURE | EXPORT_FUNCTION
internal_name     ::=  identifier |
                       string_literal    -- An operator designator
external_name     ::=  string_literal
parameter_types   ::=  NULL | ( type_mark { , type_mark } )
```

Examples:

```
procedure Matrix_Multiply(A, B: in Matrix; C: out Matrix);
pragma Export_Procedure (Matrix_Multiply);
-- External name is the string "Matrix_Multiply"
function Sin (R: Radians) return Float;
pragma Export_Function
            (Internal => Sin,
             External => "SIN_RADIANS");
-- External name is the string "SIN_RADIANS"
```

## Pragma Export_Elaboration_Procedure

The pragma Export_Elaboration_Procedure makes the elaboration procedure for a given compilation unit available to external code by defining a global symbolic name. This procedure is otherwise unnamable by the user. Its use is confined to the exceptional circumstances where an Ada module is not elaborated because it is not in the closure of the main program or the main program is not an Ada program. This pragma is not recommended for use in application programs unless the user has a thorough understanding of elaboration, runtime, and storage model considerations.

The pragma Export_Elaboration_Procedure must appear immediately following the compilation unit.

The complete syntax for this pragma is:

```
pragma_export_elaboration_procedure ::=
    PRAGMA EXPORT_ELABORATION_PROCEDURE ( EXTERNAL_NAME => external_name );

external_name ::= string_literal
```

## Pragmas Import_Object and Export_Object

Objects can be imported or exported from an Ada unit with the pragmas Import_Object and Export_Object. The pragma Import_Object causes an Ada name to reference storage declared and allocated in some external (non-Ada) object module. The pragma Export_Object provides an object declared within an Ada unit with an external symbolic name that the linker can use to allow another program to access the object. It is the responsibility of the programmer to ensure that the internal structure of the object and the assumptions made by the importing code and data structures correspond. The cross-compiler cannot check for such correspondence.

The object to be imported or exported must be a variable declared at the outermost level of a library package specification or body.

The size of the object must be static. Thus, the type of the object must be one of:

- A scalar type (or subtype)
- An array subtype with static index constraints whose component size is static
- A nondiscriminated record type or subtype

Objects of a private or limited private type can be imported or exported only into the package that declares the type.

An imported object cannot have an initial value and thus cannot be:

- Declared with the keyword *constant*
- An access type
- A record type with discriminants
- A record type whose components have default initial expressions
- A record or array whose components contain access types or task types

In addition, the object must not be in a generic unit. The external name specified must be suitable as an identifier in the assembler.

The full syntax for the pragmas Import_Object and Export_Object is:

```
object_pragma        ::=  PRAGMA object_pragma_type
                              ( [ INTERNAL => ] identifier
                                [ , [ EXTERNAL => ] string_literal ] ) ;

object_pragma_type   ::=  IMPORT_OBJECT | EXPORT_OBJECT
```

## Pragma Nickname

The pragma Nickname can be used to give a unique string name to a procedure or function in addition to its normal Ada name. This unique name can be used to distinguish among overloaded procedures or functions in the importing and exporting pragmas defined earlier.

The pragma Nickname must appear immediately following the declaration for which it is to provide a nickname. It has a single argument, the nickname, which must be a string constant. For example:

```
function Cat (L: Integer; R: String) return String;
pragma Nickname ("Int-Str-Cat");

function Cat (L: String; R: Integer) return String;
pragma Nickname ("Str-Int-Cat");

pragma Interface (Assembly, Cat);

pragma Import_Function (Internal  => Cat,
                        Nickname  => "Int-Str-Cat",
                        External  => "CAT$INT_STR_CONCAT",
                        Mechanism => (Value, Reference));

pragma Import_Function (Internal  => Cat,
                        Nickname  => "Str-Int-Cat",
                        External  => "CAT$STR_INT_CONCAT",
                        Mechanism => (Reference, Value));
```

## Pragma Suppress_All

This pragma is equivalent to the following sequence of pragmas:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Division_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Range_Check);
pragma Suppress (Storage_Check);
```

Pragma Suppress_All allows no name parameter, and it has no effect in a package specification. See LRM 11.7.3.

Note that, like pragma Suppress, pragma Suppress_All does not prevent the raising of certain exceptions. For example, numeric overflow or dividing by zero is detected by the hardware, which results in the predefined exception Numeric_Error. Refer to Chapter 7, "Runtime Organization," for more information.

Pragma Suppress_All must appear immediately within a declarative part.

## Pragma Must_Be_An_Entry

This pragma must appear in a generic formal part; it names a formal procedure previously declared in the same formal part. It is used to provide compile-time checks that: (1) the formal procedure appears in a generic formal part; (2) for each instantiation of the generic, the actual subprogram supplied for the named formal procedure is a task entry. A warning is issued if any of these checks fail.

The full syntax is:

```
entry_pragma  ::=  PRAGMA MUST_BE_AN_ENTRY
                   ( [ FORMAL_SUBPROGRAM_NAME => ]
                             ( identifier | string literal ) ) ;
```

## Pragma Must_Be_Constrained

This pragma is used in a generic formal part to indicate that formal private (and limited private) types must be constrained or need not be constrained.

This pragma allows programmers to declare explicitly how they intend to use the formals in the specification for a generic. Then the Environment can check that any instantiations of the generic that are installed before the body of the generic is installed are legal.

The pragma's syntax is:

```
pragma Must_Be_Constrained ([<cond> =>] <type_id>, ...);
```

The condition can be either yes or no.

The type identifier must be a formal private (or limited private) type defined in the same formal part as the pragma.

More than one type identifier can follow and be governed by one condition, either yes or no. If no condition precedes a type identifier, the default is yes. In the example below, Type_1, Type_2, and Type_5 will be constrained; Type_3 and Type_4 will not be.

```
pragma Must_Be_Constrained
                        (Type_1, Type_2, No => Type_3, Type_4, Yes => Type_5);
```

If the condition value of no is specified, any use in the body that requires a constrained type will be flagged as a semantic error. If yes is specified, any instantiations that contain actuals that require constrained types will be flagged with semantic errors if the actuals are not constrained.

## PREDEFINED LANGUAGE PRAGMAS (LRM ANNEX B)

The following table details the effects of predefined language pragmas.

*Predefined Pragmas*

| Pragma | Effect |
|---|---|
| Elaborate | As given in Annex B of the LRM. |
| Inline | As given in Annex B of the LRM, subject to the setting of the switch Inlining_Level. |
| Interface | Used in conjunction with pragmas Import_Procedure and Import_Function. |
| List | As given in Annex B of the LRM; evident only when the compile command is used. |
| Memory_Size | Has no effect. |
| Optimize | Has no effect. |
| Pack | Removes gaps in storage, minimizing space with possible increase in access time. See section on size of objects. |
| Page | As given in Annex B of the LRM; only evident when the compile command is used. |
| Priority | As given in Annex B of the LRM. |
| Shared | As given in Annex B of the LRM. Has an effect only for integer, enumeration, access, and fixed types. |
| Storage_Unit | Has no effect. |
| Suppress | As given in Annex B of the LRM. |
| System_Name | Has no effect. |

## IMPLEMENTATION-DEPENDENT ATTRIBUTES

The implementation-dependent attributes are as follows:

'Compiler_Version — For a name N, N'Compiler_Version is a compile-time value, a 16-character uninterpreted string that designates the compiler version used to code this Ada entity. The entity can be a program unit (package, subprogram, task, or generic), an object (variable, constant, named number, or parameter), a type or subtype (but *not* an incomplete type), or an exception. This attribute can be used for runtime detection of incompatibilities in data representation. See also 'Target_Key.

'Dope_Address — For an unconstrained array object A, A'Dope_Address is the address of the dope vector. The value is of type System.Address. This can be used for retrieving information about the object, as when reconstructing the array. See also 'Dope_Size.

'Dope_Size — For an unconstrained array object A, A'Dope_Size is the size in bits of the dope vector. The value is of type Universal_Integer. This can be used for retrieving information about the object, as when reconstructing the array. See also 'Dope_Address.

'Entry_Number — For a task entry or generic formal subprogram E, E'Entry_Number identifies the entity with a universal-integer value. This may be used by the runtime system to indicate that the entity corresponds to a process in the target and therefore requires an IPC queue.

'Mechanism — For a subprogram S with formal parameter P or the result of a function F, S'Mechanism(P) or F'Mechanism is a universal-integer value that designates the parameter-passing or function-return mechanism. This may be used by the IPC message-handling facilities to manipulate the data passed.

Currently, parameter-passing values and their meanings include:

1 parameter value is on the stack
2 parameter address is on the stack
3 parameter and dope vector address are on the stack
4 parameter address and 'Constrained datum are on the stack

Currently, function-return values and their meanings include:

11 result is returned in registers
15 address for the array result is on the stack
16 address for the result is in R0, result is on the stack
17 address for the record result is on the stack
18 address for the result is in R0, size in R1

Some details of parameter passing may change with new releases of the cross-compiler. See the release note for additional information.

'Target_Key — For a name N, N'Target_Key is a compile-time value, a 32-character uninterpreted string that designates the cross-compiler (that is, the target key) in effect when this entity was coded. This can be used for runtime detection of incompatibilities in data representation. See also 'Compiler-_Version.

'Type_Key — For a type mark T declared in a subsystem, T'Type_Key is a unique 32-character uninterpreted string. This can be used for runtime type consistency checking of message data.

## PACKAGE STANDARD (LRM ANNEX C)

Package Standard defines all the predefined identifiers in the language.

```
package Standard is

    type *Universal_Integer* is ...
    type *Universal_Real*    is ...
    type *Universal_Fixed*   is ...
    type Boolean             is (False, True);}

    type Integer             is range -2147483648 .. 2147483647;
    type Short_Short_Integer is range -128 .. 127;
    type Short_Integer       is range -32768 .. 32767;

    type Float       is digits 6  range -16#1.FFFF_FE# * 2.0 ** 127 ..
                                        16#1.FFFF_FE# * 2.0 ** 127;
    type Long_Float  is digits 15 range -16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023
                                     .. 16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023;

    type Duration    is delta 16#1.0# * 2.0 ** (-14)
                                range -16#1.0# * 2.0 ** 17 ..
                                        16#1.FFFF_FFFC# * 2.0 ** 16;

    subtype Natural  is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;

    type Character is ...

    type String is array (Positive range <>) of Character;
    pragma Pack (String);

    package Ascii is ...

    Constraint_Error : exception;
    Numeric_Error : exception;
    Storage_Error : exception;
    Tasking_Error : exception;
    Program_Error : exception;

end Standard;
```

The following table shows the default integer and floating-point types:

*Supported Integer and Floating-Point Types*

| Ada Type Name | Size |
|---|---|
| Short_Short_Integer | 8 bits |
| Short_Integer | 16 bits |
| Integer | 32 bits |
| Float | 32 bits |
| Long_Float | 64 bits |

Fixed-point types are implemented using the smallest discrete type possible; it may be 8, 16, or 32 bits. Standard.Duration is 32 bits.

## PACKAGE SYSTEM (LRM 13.7)

```
package System is

    type Address is private;

    type Name is (Mc68020_Os2000);

    System_Name : constant Name := Mc68020_Os2000;

    Storage_Unit : constant := 8;
    Memory_Size  : constant := +(2 ** 31) - 1;

    Min_Int : constant := -(2 ** 31);
    Max_Int : constant := +(2 ** 31) - 1;

    Max_Digits    : constant := 15;
    Max_Mantissa : constant := 31;
    Fine_Delta    : constant := 1.0 / (2.0 ** 31);
    Tick          : constant := 1.0 / (2.0 ** 13);

    subtype Priority is Integer range 0 .. 255;

    function To_Address (Value : Integer) return Address;
    function To_Integer (Value : Address) return Integer;

    function "+" (Left : Address; Right : Integer) return Address;
    function "+" (Left : Integer; Right : Address) return Address;
    function "-" (Left : Address; Right : Address) return Integer;
    function "-" (Left : Address; Right : Integer) return Address;
```

```
    function "<" (Left, Right : Address) return Boolean;
    function "<=" (Left, Right : Address) return Boolean;
    function ">" (Left, Right : Address) return Boolean;
    function ">=" (Left, Right : Address) return Boolean;
    --
    --  The functions above are unsigned in nature. Neither Numeric_Error
    --  nor Constraint_Error will ever be propagated by these functions.
    --
    --  Note that this implies:
    --
    --          To_Address (Integer'First) > To_Address (Integer'Last)
    --
    --  and that:
    --
    --          To_Address (0) < To_Address (-1)
    --
    --  Also, the unsigned range of Address includes values which are
    --  larger than those implied by Memory_Size.
    --

    Address_Zero : constant Address;

private
    . . .

end System;
```

## REPRESENTATION CLAUSES AND CHANGES OF REPRESENTATION

The MC68020 CDF support for representation clauses is described in this section with reference to the relevant section of the LRM. Usage of a clause that is unsupported as specified in this section or usage contrary to LRM specification will cause a semantic error unless specifically noted. Further details on the effects on specific types of objects are given in the section "Size of Objects."

### Length Clauses (LRM 13.2)

Length clauses are not supported for derived types. Length clauses are supported by the MC68020/OS-2000 CDF as follows:

- The value in a 'Size clause must be a positive static integer expression. 'Size clauses are supported for integer and enumeration types only. The value of the size attribute must be less than or equal to 32 and greater than equal to the minimum size necessary to store the largest possible value of the type.
- 'Storage_Size clauses are supported for access and task types. The value given in a Storage_Size clause can be any integer expression, and it is not required to be static.
- 'Small clauses are supported for fixed-point types. The value given in a 'Small clause must be a nonzero static real number that cannot be greater than the delta of the base type.

## Enumeration Representation Clauses (LRM 13.3)

Enumeration representation clauses are supported with the following restrictions:

- The allowable values for an enumeration clause range from (Integer'First + 1) to Integer'Last.
- Arrays indexed by enumeration types with representation clauses are not supported, unless the representation clause specifies the default representation, the representation that would have been used in the absence of the clause. This restriction on array index types is not enforced until code-generation time.

## Record Representation Clauses (LRM 13.4)

Both full and partial representation clauses are supported for both discriminated and undiscriminated records. Record component clauses are not allowed on:

- Array or record fields whose constraint involves a discriminant of the enclosing record
- Array or record fields whose constraint is not static

An error at coding time will be generated if a component clause is put on a dynamic field.

The ***static_simple_expression*** in the alignment clause part of a record representation clause (see LRM 13.4 (4)) must be a power of 2 with the following limits:

```
1 <= static_simple_expression <= 16
```

### Implementation-Dependent Components

The LRM allows for the generation of names denoting implementation-dependent components in records. For the MC68020/OS-2000 CDF, there are no such names visible to the user.

## Address Clauses (LRM 13.5)

Address clauses are not supported and will generate a semantic error if used.

## Change of Representation (LRM 13.6)

Change of representation is supported wherever it is implied by support for representation specifications. In particular, type conversions between array types or record types may cause packing or unpacking to occur; conversions between related enumeration types with different representations may result in table lookup operations.

# SIZE OF OBJECTS

This section describes the size of both scalar and composite objects. The first two subsections cover concepts of size that apply to all object types. The following subsections cover individual types. The size concepts are most important for the composite types.

## Minimum, Default, Packed, and Unpacked Sizes

The following terms are used to describe the size of objects:

- Storage unit: Smallest addressable memory unit. The size of the storage unit in bits is given by the named number System.Storage_Unit. Since the MC68020 is byte-addressable, the size of the storage unit is 8.
- Minimum size for a type: The minimum number of bits required to store the largest value of the type. For example, the minimum size of a Boolean is 1.
- Packed size for a type: The size of a component used in an array or record when a pragma Pack is in effect. This is the same as the minimum size unless modified by a 'Size clause (see "Determination of Size" below).
- Default size for a type: The smallest number of bits required to store the largest value of the type *when stored in whole storage units*. For composite types, the default sizes are multiples of 8. The possible default sizes for scalar, access, and task types are given in the following table:

***Default Sizes for Scalar, Access, and Task Types***

| Type | Sizes in Bits |
|---|---|
| Discrete | 8, 16, 32 |
| Fixed | 8, 16, 32 |
| Float | 32, 64 |
| Access | 32 |
| Task | 32 |

- Unpacked size for a type: The size of a component used in an array or record when *no* pragma Pack is in effect. This is the same as the default size unless modified by a 'Size clause (see "Determination of Size" below).
- Maximum size: The largest allowable size for a *discrete* type. For the MC68020, the maximum size is 32.

## Determination of Size

Top-level scalar and access objects are stored using their unpacked size (by top-level object we mean an object that is not a component of any array or record). Components of composite objects having neither pragma Pack nor a record representation clause are also stored using the unpacked size. Components of composite objects having pragma Pack are stored using the packed size. Fields of records having record representation clauses may be stored in any number of bits ranging from the minimum size to the default size of the field type. If a scalar or composite type component field is specified to be smaller than the default size, a filler field is introduced, and the data is left-justified. For further information, see the subsections on composite types.

'Size clauses on discrete types affect sizes by changing the packed and unpacked sizes. When there is no 'Size clause, the packed and unpacked sizes are the minimum and default sizes, respectively. 'Size clauses with values outside the minimum and maximum sizes cause a semantic error. Within that range, there are two cases depending on the value specified by the clause:

- *Value* <= Default Size: The packed size is set equal to *value*. The unpacked size is not affected.
- *Value* > Default Size: For integer types, the 'Size clause will cause a semantic error.

***Size Examples for MC68020 Target***

| Type Declaration of the Example | Minimum Size | Default Size | Maximum Size |
|---|---|---|---|
| Integer | 32 | 32 | 32 |
| Boolean | 1 | 8 | 32 |
| Float | 32 | 32 | 32 |
| type Byte is range 0 .. 255 | 8 | 16 | 32 |
| type Primary is (Red, Blue, Yellow) | 2 | 8 | 32 |
| type X is (Normal, Read_Error, Write_Error) for X use (7, 15, 31) | 5 | 8 | 32 |
| type Ary is array (1 .. 100) of Boolean | 100 | 800 | n/a |

## Integer Types

An integer type with a range constraint has a default size the same as that of the smallest integer type defined in package Standard that will hold its range. For example, consider the following type declaration:

```
type Byte is range 0..255;
```

Type Byte will have a minimum size of 8 and a default size of 16. It has a default size of 16 because the smallest type from which Byte can be derived is Short_Integer. (Short_Short_Integer, which has a size of 8, does not include values greater than 127.)

The 'Size clause is supported for integer types that are not derived types. The effect of a 'Size clause on minimum size is shown in the following example. Consider:

```
type Byte is range 0 .. 255;
for Byte'Size use n;
```

where *n* is a static integer expression. The following table shows the effect of *n* on the packed and unpacked sizes.

*Example of Effect of 'Size Clauses*

| 'Size Clause | Packed Size | Unpacked Size |
|---|---|---|
| No 'Size clause | 8 | 16 |
| Use 8 | 8 | 16 |
| Use 12 | 12 | 16 |
| Use 16 | 16 | 16 |

## Enumeration Types

For an enumeration type with $n$ elements, the default internal integer representation range is 0 .. $n$–1. The maximum number of elements that may be declared for any one enumeration type depends on the total number of characters in the images of the enumeration literals. Let $L$ be the total number of characters of the $n$ elements. Then $L$ and $n$ must satisfy the following inequality: $2n + 4 + L < 2^{16}$.

'Size clauses are not supported for derived types. An enumeration clause on a derived type is supported only if the parent type has no enumeration clause. Derived types inherit the parent type's minimum and default sizes, as well as the internal integer representation unless, an enumeration clause applies.

For predefined type Character, the value returned by the 'Size attribute is 8, and the minimum size is 8. User-defined Character types behave like ordinary enumeration types and may have a minimum size of less than 8.

The 'Size clause is supported for enumeration types that are not derived types. The effect of a 'Size clause on representation is shown in the following example. Consider:

```
type Response is (No, Maybe, Yes);
for Response'Size use n;
```

where $n$ is a static integer expression. The following table lists the packed and unpacked sizes for different values of $n$.

*Example*

| 'Size Clause | Packed Size | Unpacked Size |
|---|---|---|
| No 'Size clause | 2 | 8 |
| Use 4 | 4 | 8 |
| Use 12 | 12 | 16 |
| Use 16 | 16 | 16 |
| Use 20 | 20 | 32 |
| Use 32 | 32 | 32 |

## Floating-Point Types

The internal representations for floating-point types are the 32-bit and 64-bit floating-point representations as outlined by the MC68020 architecture. The 'Size clause is not supported for floating-point types; a semantic error will be generated if one is used.

## Fixed-Point Types

Fixed-point numbers are represented internally as integers. The integer representation is computed by scaling (dividing) the fixed-point number by the actual small implied by the fixed-point type declaration. The values that are exactly representable are those that are precise multiples of the actual small; numbers between those values are represented by the closest exact multiple. For example, in the declaration:

```
type fix is delta 0.01 range -10.0 .. 10.0;
```

the integer value used to represent the lower bound of the type is –10.0 / (1/128), or –1280, since the actual small is 1/128. In the example:

```
type fix is delta 0.01 range -10.6 .. 10.6;
```

the integer value used to represent the lower bound of the type is –1357, which is the closest exact multiple of the actual small.

The size of the representation is 32, 16, or 8 bits; the compiler chooses the smallest of these that can represent all of the safe numbers of the fixed-point type.

## Access Types and Task Types

Access and task objects have a size of 32 bits. The 'Storage_Size length clause is allowed for access and task types. The value given in a Storage_Size clause may be any integer expression, and it is not required to be static. Static expressions larger than Integer'Last will generate compilation warnings; however, a Numeric_Error exception will be raised at run time. For access types, a 'Storage_Size clause is used to specify the size of the access type's collection. If a 'Storage_Size clause has been applied to an access type, the collection is nonextensible. For task types, the clause determines the stack size.

A value (either static or not) of 0 is allowed; in this case, no collection or task stack space will be allocated, and Storage_Error will be raised at run time if any attempt is made to allocate or deallocate from the collection or activate the task. Negative values are also allowed by the CDF; however, this will generate a Storage_Error exception when the type is elaborated even if no attempt is made to allocate or deallocate objects belonging to the collection.

## Composite Types

The size of a composite type depends on whether or not it is packed. It is packed if and only if there is a pragma Pack affecting the type.

Other factors affecting size are the presence of:

- A record representation clause
- Alignment filler
- Tail filler

These factors, which mostly affect records, are dealt with first in the following subsections. Once their effects are understood, the differences between packed and unpacked representations are fairly simple. Later subsections deal with packed and unpacked representation and pragma Pack.

### Using a Record Representation Clause

If any components or parts of components are covered by a record representation clause, then that clause controls, whether the composite type is packed or not.

If a record representation clause is used, some fields of a record may not be determined by that clause. Such fields may be influenced by packing. Also, these fields are placed after all those that are specified by the clause. In particular, for a discriminant not governed by a clause, the discriminant field is placed after all the governed fields. Since the discriminant field must then be placed after the largest possible field, there will be no space savings gained by using the constrained subtype of the discriminated record.

### Alignment Filler

Alignment filler may be present in a composite type when it is a record or is an array containing records. In the absence of a record representation clause, the compiler reorders record fields in an attempt to maintain alignment and reduce the need for alignment filler fields. Nonetheless, sometimes it is necessary to introduce alignment filler fields to maintain alignment. For example, in an unpacked array of records consisting of a character and an integer, the integers would be aligned on longword boundaries by introducing an alignment filler of three bytes between array components.

### Tail Filler

The last storage unit of the composite type may contain some unused bits; these bits, called *tail filler*, are zeroed when an object is elaborated so that block comparisons can be performed. Tail filler does *not* contribute to the size of the composite type as computed by the 'Size attribute.

### Unpacked Composite Types

When a composite type is unpacked, each component is stored in the same size it would have as a top-level object. For scalars, that is its unpacked size. Components covered by a record representation clause obey that clause. For components of composite type, any packing within the component will be retained.

### Packed Composite Types

When a composite type is packed, scalar components are stored in packed size. Packed and unpacked size can differ only for integer and enumeration types. Tail filler between compo-

nents is eliminated; for arrays, alignment filler within components is eliminated; for records, it is not.

### Limitations on the Effect of Pragma Pack

- The packed size of discrete types that have a minimum size of 27, 29, 30, or 31 bits is set to 32.
- When an object with a pragma Pack on it is used as a component or subcomponent of a composite type, applying a pragma Pack to the composite type will have no effect on the composite.

### Change of Representation for Packed Composite Types

Change of representation for packed composite types may cause extra code to be generated to do packing and unpacking conversion. For example:

```
type A is array (1..10) of Boolean;  -- Size of 80 bits
type B is new A;
pragma Pack (B);                     -- Size of 10 bits

X : B;
Y : A;
X := B(Y);  -- Extra code will be generated here
            -- to convert from type B to type A
```

## OTHER IMPLEMENTATION-DEPENDENT FEATURES

## Machine Code (LRM 13.8)

Machine-code insertions are not supported at this time.

## Unchecked Storage Deallocation (LRM 13.10.1)

Unchecked storage deallocation is implemented by the Unchecked_Deallocation generic function defined by the LRM. This procedure can be instantiated with an object type and its access type, resulting in a procedure that deallocates the object's storage. Objects of any type may be deallocated.

The storage reserved for the entire collection associated with an access type is reclaimed when the program exits the scope in which the access type is declared. Placing an access type declaration within a block can be a useful implementation strategy when conservation of memory is necessary.

Erroneous use of dangling references may be detected in certain cases. When detected, the Storage_Error exception is raised. Deallocation of objects that were not created through allocation (that is, through Unchecked_Conversion) may also be detected in certain cases, also raising Storage_Error.

## Unchecked Type Conversion (LRM 13.10.2)

Unchecked conversion is implemented by the Unchecked_Conversion generic function defined by the LRM. This function can be instantiated with Source and Target types, resulting in a function that converts source data values into target data values.

Unchecked conversion moves storage units from the source object to the target object sequentially, starting with the lowest address. Transfer continues until the source object is exhausted or the target object runs out of room. If the target is larger than the source, the remaining bits are undefined.

### Restrictions on Unchecked Type Conversion

- The target type of an unchecked conversion cannot be an unconstrained array type or an unconstrained discriminated type without default discriminants.
- Internal consistency among components of the target type is not guaranteed. Discriminant components may contain illegal values or be inconsistent with the use of those discriminants elsewhere in the type representation.

# CHARACTERISTICS OF I/O PACKAGES

This section specifies the implementation-dependent characteristics of the I/O packages Sequential_Io, Direct_Io, Text_Io, and Io_Exceptions. These packages are located in the library !Targets.Mc68020_Os2000.Io. Nondependent characteristics are as given in Chapter 14 of the LRM. Each section below cites the relevant section in Chapter 14.

The package Os2000_Io is provided as an interface to many of the OS-2000 I/O system calls. Os2000_Io is in the library !Targets.Mc68020_Os2000.Target_Interface.

Package Low_Level_Io is not provided for the Mc68020_Os2000 target.

## External Files and File Objects (LRM 14.1)

An external file is identified by a Name and further characterized by a Form parameter. The allowable strings for the Name are the legal filenames and full or relative path lists accepted by OS-2000. See the *OS-9/68000 Operating System User's Manual* for details. There are no recognized values for the Form parameter.

If a main program completes without closing some Text_Io file, any data output to the file after the last line or page terminator will not be included in the associated external file.

Input and output are erroneous for access types.

## Sequential and Direct Files

This section deals with implementation-dependent features associated with the packages Sequential_Io and Direct_Io and the file types *sequential access* and *direct access*.

### File Management (LRM 14.2.1)

The Use_Error exception is raised in the following situations:

- By procedure Create if an external file with the specified Name already exists.
- By procedure Open if the executing process does not have correct access rights for the external file.
- By procedure Open if another process has already opened the external file for nonsharable use.
- By either procedure Create or Open if accessing the external file would exceed the OS-2000 limit on the number of open files for the executing process.
- By procedure Open if the external file is currently opened by another process with mode Out or Inout.

### Sequential Input/Output (LRM 14.2.2)

For the Read procedure of Sequential_Io, the Data_Error exception is raised only when the size of the data read from the file is greater than the size of the *out* parameter Item.

### Direct Input/Output (LRM 14.2.4)

Package Direct_Io may not be instantiated with any type that is either an unconstrained array type or a discriminated record type without default discriminants. A semantic error is reported when attempting to install any unit that contains an instantiation in which the actual type is such a forbidden type.

For the Read procedure of Direct_Io, there is no check performed to ensure that the data read from the file can be interpreted as a value of the Element_Type.

### Specification of Package Direct_Io (LRM 14.2.5)

The declaration of the type Count in package Direct_Io is:

```
type Count is new Integer range 0 .. Integer'Last / Element_Type'Size;
```

where Element_Type is the generic formal type parameter.

## Text Input/Output (LRM 14.3)

The Text_Io default input and output files are associated with the OS-2000 standard input and standard output paths, respectively. If a program is initiated without any of the standard input, standard output, or standard error output, then those paths are opened for device/nil. The terminators used by Text_Io are Ascii.Cr for the line terminator, and the sequence Ascii.Ff, Ascii.Cr for the page terminator, except for terminators at the end of file, which are implicit and not represented by any characters.

### File Management (LRM 14.3.1)

The Text_Io function Name raises Use_Error when called with either Text_Io.Standard_Input or Text_Io.Standard_Output as the actual parameter. This implementation for Name was chosen because OS-2000 does not provide a uniform mechanism for obtaining a string name for the external file associated with a path.

### Specification of Package Text_Io (LRM 14.3.10)

The declaration of the type Count in Text_Io is:

```
type Count is range 0 .. 1_000_000_000;
```

The declaration of the subtype Field in Text_Io is:

```
subtype Field is Integer range 0 .. Integer'Last;
```

## Exceptions in I/O (LRM 14.4)

The exceptions raised in input/output operations are:

- The exceptions as specified in LRM 14.4.
- The Use_Error exception as specified in the two file management subsections above.
- The Device_Error exception, raised for any input/output operation that performs an OS-2000 I/O system call returning as status the error code E$Read or E$Write.
- The Device_Error exception, raised if the status returned from any OS-2000 I/O call is not among the error codes E$Pnnf, E$Bpnam, E$Bmode, E$Eof, E$Pthful, E$Fna, E$Share, E$Bpnum, E$Cef, E$Read, or E$Write.

# Index

RATIONAL

## READER'S COMMENTS

**Note:** This form is for documentation comments only. You also can submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____

_____

_____

_____

_____

_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____

_____

_____

_____

Indicate any additions or changes you would like to see in the index.

_____

_____

_____

_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____

Company _____

Address _____

City _____ State _____ ZIP Code _____

**Please return this form to:**

**Publications Department**
**Rational**
**3320 Scott Boulevard**
**Santa Clara, CA 95054-3197**

RATIONAL

## READER'S COMMENTS

**Note:** This form is for documentation comments only. You also can submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

**Please return this form to:**

**Publications Department**
**Rational**
**3320 Scott Boulevard**
**Santa Clara, CA 95054-3197**