Rational M68K/OS-9 Cross-Development Facility

Document Control Number: 8027A
Rev. 0.0, February 1988 (1_2_0)

This document subject to change without notice.

Rational
1501 Salado Drive
Mountain View, California 94043

Preface

The *Rational M68K/OS-9 Cross-Development Facility* manual (prerelease version) contains introductory material and descriptions of the components and features of the Rational M68K/OS-9 Cross-Development Facility. This manual is divided into sections with identifying tabs. There is a master table of contents for the manual.

The *Rational M68K/OS-9 Cross-Development Facility* manual assumes familiarity with Ada, the M68K/OS-9 instruction set, the OS-9/68000 operating system, and the Rational Environment. It also assumes familiarity with the user interface of the Rational Editor. For more information, consult the *Rational Environment Reference Manual*, the *Rational Environment Basic Operations*, the *OS-9/68000 Operating System Technical Manual*, and the *OS-9/68000 Operating System User's Manual*.

The information in this manual is preliminary and is subject to change with newer releases of the Rational M68K/OS-9 Cross-Development Facility.

If you have any comments about the organization or content of this manual, please address your comments to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043

Contents

1. Key Concepts

It often is necessary to develop software on one class of machine (for example, the Rational® R1000®) and execute the software on another machine (for example, D85 hardware). Two principal kinds of programs require this approach:

- Programs that are to run on small, embedded targets. These targets are powerful enough to run the end application but are not capable of supporting a complete software-development environment. That is, they are too small to support an operating system but contain a kernel that is large enough to run the application. Examples of this class of target are:

  — MIL-STD-1750A architecture

  — Bare Motorola® M68000 (M68K) microprocessor family

  Each of these requires a separate host for the development, debugging, testing, and integration of the software.

- Programs that are designed to run on various computer hardware architecture and operating systems. Although the computers may be capable of developing the software, their software-development environments may not be sufficiently advanced to allow the code to be developed on schedule. Perhaps the software is to be developed to run on a variety of different targets. Developing it on each target could lead to different software being run on each target. Using one machine as the development machine and the other machines as the target machines is a good solution.

A cross-development facility (CDF) provides the mechanism for developing programs to run on embedded systems and generating programs that will run on machines with their own operating systems. The programs are developed on the universal host using all the software-engineering support features provided by the host machine. However, instead of generating host-specific executable modules designed to run on the host, the host generates target-specific code that can be executed on the target machine. The code produced on the host can be targeted to many different target machines. For example, with care taken to ensure portability, code developed on a Rational R1000 can be ported to a MIL-STD-1750A architecture, to a bare M68K architecture, to an M68K architecture that has its own operating system, and to a VAX™ architecture. Although the executable modules that actually run on each target machine are different, the source code developed on the host machine is the same for all of them. If a bug must be fixed, it can be fixed on the universal host and then the change can be ported to each target machine. Each target machine will still be running the same program as the others.

Different strategies can be employed in 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 universal host, the object modules are linked into an executable module on the universal host, and the executable module is ported to the target machine on which it is to be executed. (See Figure 1-1.)

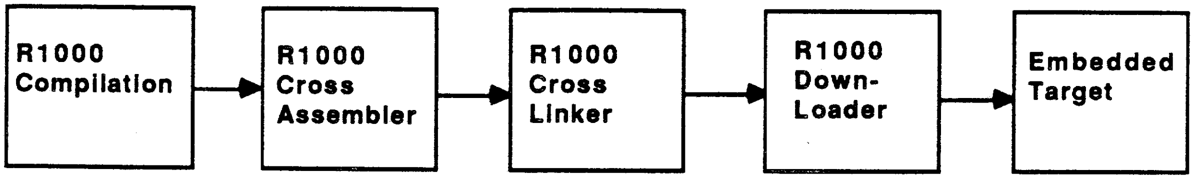

Figure 1-1 CDF Used with Embedded Target

- For architectures with their own operating systems, the Ada source code is generated on the universal host, the source code is compiled into object modules on the universal host, the object modules are linked into an executable module on the universal host, and the executable module is ported to the target machine on which it is to be executed. (See Figure 1-2.)

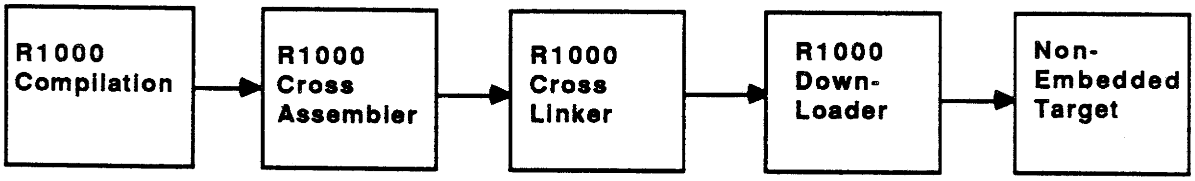

Figure 1-2 CDF Used with Nonembedded Target and R1000 Cross-Assembler and Linker

- For architectures with their own operating systems, the Ada source code is generated on the universal host, the object modules are created using facilities provided by the target machine, the object modules are linked into an executable module using the linker provided by the target machine, and the executable module is executed on the target machine. (See Figure 1-3.)

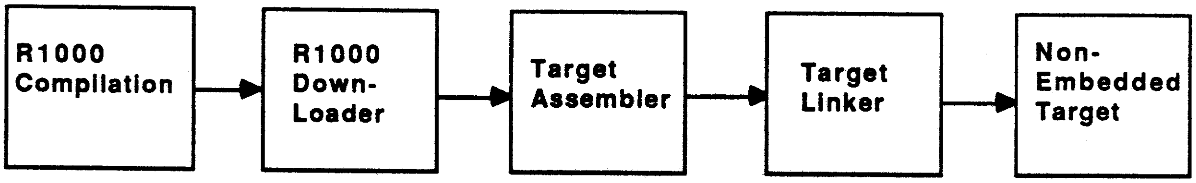

Figure 1-3 CDF Used with Nonembedded Target and Target's Assembler and Linker

For any of these strategies, a critical component of the CDF is the software that ports the code from the universal host to the target machine, the downloader.

The Rational Cross-Development Facilities provide solutions to all of the above strategies. These facilities generate target-specific assembly source code. This assembly source code can be assembled on the host cross-development assembler and linked on the host cross-development linker, and then the executable module can be downloaded to the target. Alternatively, the assembly code can be downloaded to the target; then assembly and linking occur using the target's assembler and linker.

The following chapters describe in detail the Rational solution to developing code on the R1000 to be executed on the M68K architecture.

2. Rational M68K/OS-9 Overview

The Rational M68K/OS-9 Cross-Development Facility (CDF) provides you, the developer using the Rational R1000 Development System, the ability to write, compile, assemble, link, and debug Ada programs that will be targeted for execution on D85 hardware. The R1000 acts as a universal host on which you can create application programs using the facilities provided by the R1000 to write Ada code. You then use the CDF to cross-compile the code into M68K/OS-9 instructions, assemble the code to produce an object module, and then link this object module with other object modules (for example, run-time modules from the run-time library) to produce an executable module. The executable module is converted to the OS-9 object-module format that is transferred to the D85 hardware. The end result of this process is the creation of code that executes on the D85 hardware. The code can be executed and debugged on the D85 hardware.

2.1. Capabilities

Using the features provided in this release, you can accomplish the following:

- Compile, assemble, and link application programs targeted to M68K/OS-9 processors.
- Write your own assembly-language programs, assemble them, and then link them with other assembled modules and/or assembled Ada code into executable modules.
- Debug your Ada program using the M68K/OS-9 cross-debugger.

2.1.1. Compilation Mode

This section discusses two compilation modes:

- R1000
- Motorola_68k

To develop applications targeted for the M68K/OS-9, you need to understand both the R1000 and the Motorola_68k compilation modes. These modes differ in certain specific ways.

2.1.1.1. R1000 Compilation Mode

In the R1000 compilation mode, an Ada unit can exist in one of three states:

- Source
- Installed
- Coded

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 Rational Environment™ provides facilities for syntactic and semantic checking of this code.

The unit in the source state can be promoted to a higher state, the *installed* state. The code in this unit is syntactically and semantically correct. If this unit depends on other Ada units, the Environment ensures that these units also are installed. Incremental operations can be performed on units in the installed state.

The installed unit can be promoted to a *coded* unit. The unit can now be executed. Some incremental operations also can occur in the coded state. If other units are required to execute your unit (that is, they are *with*ed by your units), the Environment manages the unit dependencies within your program, ensuring that all the units are in the coded state. If the unit does not exist or cannot be coded, the Environment provides an appropriate error message.

When executing your Ada program, you must know which unit is the main unit. A Main pragma optionally may be added to the unit to indicate that it is a main unit.

Your units will execute on the R1000 hardware using the R1000 tools available (for example, the debugger).

In summary, the major features of compilation on the R1000 are the following:

- The Ada unit exists in a source, installed, or coded state. There are no separate source and object files.
- The coded state of an Ada unit 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 Main pragma is not required to specify which Ada unit is the main unit.
- The coded units execute on R1000 hardware.

2.1.1.2. Motorola_68k Compilation Mode

In the Motorola_68k compilation mode, the Ada unit also exists in one of three states:

- Source
- Installed
- Coded

The first two states are similar to the states in the R1000 mode; any operations that you can perform on the R1000 objects, you also can perform on the Motorola_68k objects.

The coded state, however, is somewhat different. By default, the output of the coding process is a relocatable object module for each Ada library unit. For program execution, these relocatable object modules must be linked into an executable module.

For the linker to produce an executable module for the program, the spec or body of the main Ada unit must contain a Main pragma. This pragma tells the linker that the unit is 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 M68K linker is invoked automatically, and it links all of the program's object modules along with necessary modules from the Ada run-time library to produce an executable module (and a link map describing the program layout).

The executable module produced by the linker must be converted from the R1000 object-module format to the OS-9 object-module format. The converted file must be transferred to the D85 hardware, where it can be run either directly using OS-9 operating-system commands or indirectly using the R1000-hosted M68K cross-debugger. The M68k cross-debugger has the same user interface that is used for debugging R1000 programs, but it provides additional specific M68K/OS-9 debugging operations (for example, machine instruction stepping).

In summary, the major features of compilation on the M68K CDF are the following:

- 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 Main pragma is required to specify what Ada unit is the main unit. When the main unit is promoted, it invokes the linker to generate an executable module.
- The executable file is converted to the OS-9 object-module format and transferred to the D85 hardware.
- The executable module can be executed directly on D85 hardware using OS-9 operating-system commands or indirectly using the M68K cross-debugger.

2.1.2. Worlds

Worlds are components of the R1000 library structure that encapsulate the library's name space, thus providing physical and logical separation of library contents from enclosing libraries. Within a world, there are internal links between Ada units. To import Ada units residing outside a library, you must create external links to these units.

2.1.2.1. Defining a World

The Rational Environment provides facilities for defining a model world that can be used to initialize newly created worlds. Models typically contain:

- A set of external links

- Ada units

- Objects such as switch files

Instead of creating your own model world, you can use default models provided by the Environment. The models of interest for the M68K/OS-9 CDF are:

- R1000: Includes links for the R1000 facilities.

- R1000_Portable: Includes links only for Ada-specified standard facilities to ensure portability.

- Motorola_68k: Includes links for M68K/OS-9 facilities and additional Rational-provided facilities for the M68K/OS-9.

- Motorola_68k_Portable: Includes links for Ada-specified standard M68K/OS-9 facilities to ensure portability.

2.1.2.2. Target Keys

Each world has a target key associated with it. The target key specifies what target-dependent compiler is invoked when you enter a compilation command (such as Compilation.Make or Promote), as well as what target-specific cross-debugger operations are available. The compilers have different code generators and produce target-specific code. The target keys are set by commands that are discussed in a later chapter. The two target keys of interest here are:

- R1000 (the default)

- Motorola_68k

The R1000 target key selects:

- R1000-dependent semantic checking

- R1000 code generator

- R1000 debugger

The Motorola_68k target key selects:

- M68K/OS-9-dependent semantic checking
- M68K/OS-9 code generator
- M68K/OS-9 cross-debugger

2.2. Major Components

The major components of the M68K/OS-9 CDF present in this release are:

- Cross-compiler
- Cross-assembler
- Cross-linker
- Object-module converter
- Cross-debugger
- Run-time library
- OS-9 file-transfer software

3. Using the M68K/OS-9 Cross-Development Facility

This chapter provides the necessary steps that you will need to create, compile, assemble, link, execute, and debug your programs in the M68K/OS-9 environment. The following steps will be discussed in the order required to execute a program:

- Preparing the Motorola_68k environment
- Establishing a Motorola_68k library-switch file
- Creating Ada units
- Compiling, assembling, and linking
- Converting to OS-9 object-module format
- Transferring the executable file to the hardware
- Executing and debugging on the hardware

3.1. User Scenario

The following scenario illustrates the use of the M68K/OS-9 Cross-Development Facility (CDF) to develop executable modules that will run on D85 hardware:

1. Develop the source units on the R1000 in an R1000 world. These units can be developed and tested using all the facilities provided by the Rational Environment. The tested code then can be ported to a program library on the R1000 that is a Motorola_68k world. You also can develop M68K assembly code and keep it in a Motorola_68k world on the R1000.

2. Promote the Ada units to the coded state by pressing [Code (This World)].

   - The appropriate compiler is selected automatically, and it generates M68K assembly source code.
   - The assembler automatically assembles this source code into relocatable object modules.
   - The linker automatically links the object modules and any required run-time modules into an executable module.

3. Assemble user-developed assembly source code.

   - The Ada code must have the appropriate pragmas to deal with the external assembly code.
   - The assembler must be invoked explicitly by the M68k.Assemble command.
   - The linker-command file must be modified to include the external assembly source.

4. Convert the executable module from R1000 object-module format to OS-9 object-module format.

5. Transfer the converted executable module to the D85 hardware.

6. Run the OS-9 executable module on the D85 hardware by invoking the OS-9 operating-system commands through an OS-9 console or by invoking the M68K cross-debugger.

Figure 3-1 shows a possible user scenario.

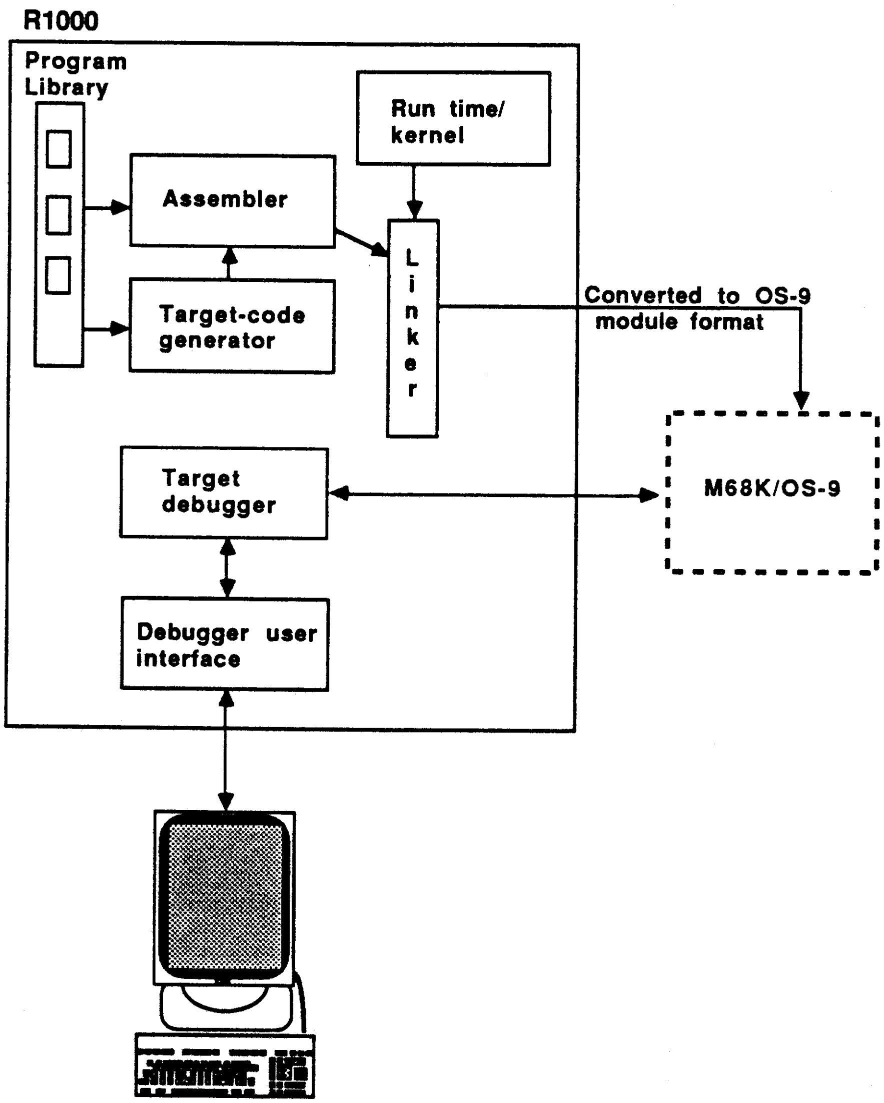

Figure 3-1 Possible User Scenario

3.2. Preparing a Motorola_68k Environment

Development of programs targeted to D85 hardware must be done within subsystem views or worlds created for M68K development. Failure to correctly specify a Motorola_68k view or world will prevent you from compiling programs using the M68K CDF.

3.2.1. Using Motorola_68k Subsystem Views

You can use subsystem views to compile, assemble, and link your Ada units and use the facilities provided by configuration management and version control (CMVC) to manage your project development. It is assumed that you are already familiar with Rational Subsystems™; therefore, only the specifics required to operate with the M68K CDF will be discussed. If you require more information on subsystems, consult the *Rational Environment Basic Operations* manual.

3.2.1.1. Creating a Motorola_68k Path from an R1000 Path

You probably will do most of your development in an R1000 view and then accept these changes into a Motorola_68k view. You can accomplish this by creating a path between an R1000 view and a Motorola_68k view. However, the views do not have to be the same kind—the R1000 can be a spec/load view.

To create a Motorola_68k path from an R1000 path:

1. From the R1000 view from which you want to create the path, create a Command window.

2. Enter `Cmvc.Make_Path` and press [Complete].

3. Set the From_Path parameter to the R1000 view from which you want to create the path (for example, `Rev1_Working`). You could have used the default.

4. Set the New_Path_Name parameter to the name of the Motorola_68k view (for example, `Motorola_68k_Working`).

5. Set the Model parameter to `Motorola_68k`.

6. Set the Create_Load_View parameter to `True`.

7. Press [Promote].

For example, the following command establishes a Motorola_68k_Working path that is joined to Rev_1_Working:

```
Cmvc.Make_Path (From_Path => "Rev1_Working",
                New_Path_Name => "Motorola_68k_Working",
                View_To_Modify => "",
                View_To_Import => "<INHERIT_IMPORTS>",
                Only_Change_Imports => True,
                Create_Load_View => True,
                Model => "Motorola_68k",
                Join_Paths => True,
                Create_Combined_View => False,
                Remake_Demoted_Units => True,
                Goal => Compilation.Coded,
                Comments => "",
                Work_Order => "<DEFAULT>",
                Volume => 0,
                Response => "<PROFILE>");
```

You can use CMVC to check out units in one view, modify them, and then check in the units. The other view can check out and check in the same units. You can accept changes made in one view into the other.

There may be some units, however, for which changes made in one view could be incompatible with the same units in the other view (for example, target-dependent units). You would not want changes made in one view to be reflected in the other view. To prevent changes made in one view from being accepted into another view, sever the path between the involved units using the following steps:

1. From the view that contains the units you want to sever, create a Command window.

2. Enter `Cmvc.Sever` and press [Complete].

3. Set the What_Objects parameter to the name of the unit you want to sever.

4. Press [Promote].

For example, the following command severs the selected objects:

```
Cmvc.Sever (What_Objects => "<SELECTION>",
            New_Reservation_Token_Name => "<AUTO_GENERATE>",
            Comments => "",
            Work_Order => "<DEFAULT>",
            Response => "<PROFILE>");
```

You can now make changes to the units in one view without affecting the units in the other view.

3.2.2. Using Motorola_68k Worlds

You have seen how to create subsystem views so that you can compile, assemble, and link Ada units using the CDF. An alternative method is to create a Motorola_68k world and set the Motorola_68k target key. For the M68K compiler to be invoked, the unit being compiled must reside in a Motorola_68k world. You create this world using a two-step process: create the world and set the target key.

3.2.2.1. Creating a Motorola_68k World

To create a Motorola_68k world initialized with the proper links:

1. From the library that will contain the Motorola_68k world, press [Create World].

2. Enter the world name.

3. Rational supplies two model worlds for the M68K:

   - !Model.Motorola_68k: Includes links for Motorola_68k facilities, including nonstandard packages provided by Rational.

   - !Model.Motorola_68k_Portable: Includes links for Ada-specified standard Motorola_68k facilities to ensure portability.

   Enter `!Model.Motorola_68k` (or `!Model.Motorola_68k_Portable`) following the `Model` prompt.

4. Press [Promote].

For example, the following command creates a world called Heinze_Motorola_68k_Programs:

```
Library.Create_World
   (Name => "Heinze_Motorola_68k_Programs",
    Kind => Library.World,
    Vol => Library.Nil,
    Model => "!Model.Motorola_68k",
    Response => "<Profile>");
```

3.2.2.2. Setting the Target Key

After creating a Motorola_68k world, you must associate the appropriate Motorola_68k target key with it to ensure that the proper cross-compiler and cross-debugger are selected.

To set the target key:

1. From the newly created world, create a Command window.

2. Enter `Compilation.Set_Target_Key` and press [Complete].

3. Enter `Motorola_68k` following `The_Key` prompt.

4. Press [Promote].

For example, the following command sets the target key to Motorola_68k:

```
Compilation.Set_Target_Key
     (The_Key => "Motorola_68k",
      To_World => "<Image>",
      Response => "<Profile>"),
```

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

```
= !USERS.YOUR_NAME.WORLD_NAME (library)          Motorola_68k WORLD
```

3.3. Library Switches Specific to the M68K/OS-9 Cross-Development Facility

When a Motorola_68k world is created, a library-switch file must be associated with that world if you want to control some of the behavior of the M68K 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 will already exist. You can modify these switches if necessary.

3.3.1. Creating the Switch File

Note that 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 Motorola_68k world by one of the following methods:

- *Method 1*

  1. From the new Motorola_68k world, create a Command window.

  2. Enter `Switches.Edit` and press [Complete].

  A switch file called Library_Switches will be associated with the Motorola_68k world.

- *Method 2*

1. From the new Motorola_68k world, create a Command window.
2. Enter `Switches.Create` and press [Complete].
3. Enter the name you want to give the switch file at the `File` prompt.
4. Press [Promote].

   For example, the following command creates a switch file called Heinze_Motorola_68k_Switches:

```
Switches.Create (File => "Heinze_Motorola_68k_Switches",
                 Category => 'L',
                 Response => "<PROFILE>");
```

5. From a Command window, enter `Switches.Associate` and press [Complete].
6. Enter the switch filename at the `File` prompt.
7. Press [Promote].

   For example, the following command associates the Heinze_Motorola_68k_Switches with the library:

```
Switches.Associate (File => "Heinze_Motorola_68k_Switches",
                    Library => "<IMAGE>",
                    Response => "<PROFILE>");
```

You can now display and/or edit the Motorola_68k library switches as necessary.

3.3.2. Cross-Compiler Switches

The switches of interest for the M68K/OS-9 cross-compiler are the Cross_Cg processor switches. The Cross_Cg switches provided are:

- Asm_Source: Takes a Boolean value; controls the 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 Main pragma. The default value is true.

- Debugging_Level: Controls the amount of 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 certain optimizations are inhibited. ("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.

- 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 an integer value in the range 0 .. 4; controls the amount of optimization performed during code generation (see Chapter 4, "M68K/OS-9 Cross-Compiler," for a full discussion of the optimizations). The possible values are:

  0: Constant folding
  Algebraic transformations
  Context determination

  1: Peephole optimizations
  Elimination of common subexpressions within basic blocks
  (Plus the optimizations found in level 0)

  2: Lifetime analysis for variables
  Value tracking
  Elimination of unreferenced value assignments
  Elimination of dead variables
  Construction of flow and call graphs
  Elimination of dead code
  Mapping of local variables onto registers
  Register targeting
  Determination of evaluation ordering
  Short-circuit evaluation
  Constraint-check elimination
  (Plus the optimizations found in levels 0 and 1)

  3: Loop-strength reduction
  (Plus the optimizations found in levels 0, 1, and 2)

4: Inline expansion
(Plus the optimizations found in levels 0, 1, 2, and 3)

- Suppress_All_Checks: Takes a Boolean value; when the switch is set to true, it has the same effect as a Suppress_All pragma at the beginning of each Ada unit in the library. The default value is false.

In addition to the Cross_Cg switches, the following two FTP switches are of interest because they are used in transferring the converted executable files to the D85 hardware and are used by the M68K cross-debugger to select a remote machine and a remote directory on the D85 hardware:

- Remote_Directory: Takes a string value; specifies the name of the directory on the D85 hardware that will receive the executable file.

- Remote_Machine: Takes a string value; specifies the name of the D85 hardware that receives the executable file.

3.4. Creating Ada Units

Now that you have created a subsystem or Motorola_68k 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 Motorola_68k view or world, or you can create them in an R1000 view or world and then port them to the Motorola_68k view or world.

3.4.1. Creating Ada Units in a Subsystem or a Motorola_68k World

Creating units in a Motorola_68k 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. You use the [Format] key for syntax checking, syntactical completion, and pretty-printing. You use the [Semanticize] key for incremental checking of Ada semantics.

There is one requirement in creating Ada units in the Motorola_68k view or world: you must have a Main pragma at the end of the main unit's specification or body. This is required to invoke the linker.

3.4.2. Porting R1000-Developed Ada Units to a Motorola_68k 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, port them to an M68K. Care must be taken not to *with* any packages that are specific to the Rational Environment, because these will not function on the M68K cross-compiler.

The programs can be transferred between worlds using the Library.Copy command. With this procedure, you also can copy the links into the Motorola_68k world, but because these links are to R1000 library units rather than equivalent Motorola_68k library units, you should set the Copy_Links parameter to false and explicitly create links between Motorola_68k worlds.

To transfer units from an R1000 to a Motorola_68k world:

1. From a Command window, enter `Library.Copy` and press [Complete].

2. Enter the name of the R1000 world at the `From` prompt.

3. Enter the pathname of the Motorola_68k world at the `To` prompt.

4. Change the Copy_Links parameter to `False`.

5. Press [Promote].

For example, the following command transfers the objects to the Motorola_68k world, where they will be in the source state:

```
    Library.Copy
        (From => "!Users.Wjh.R1000_Directory",
         To => "!!Jazmo!Users.Wjh.Motorola_68k _Directory",
         Recursive => True,
         Response => "<Profile>",
         Copy_Links => False,
         Options => "");
```

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

3.4.3. Porting R1000-Developed Ada Units to a Motorola_68k Path

If you are using subsystems, you can create, debug, and functionally test your programs in an R1000 path. Once the programs are executing properly, you can use the CMVC facilities to automatically port the units to a Motorola_68k path using the Cmvc.Accept_Changes command.

Note that, because you often do not use a Main pragma when you create Ada units in an R1000 view or world, you must make sure that there is a Main pragma on the main Ada unit. You can enter the Main pragma at the main unit's specification or body before you port the units to the Motorola_68k view or world, or you can enter it into the main unit after you have ported the units.

3.5. Compiling, Assembling, and Linking Ada Programs

Now that you have Ada units in your Motorola_68k 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.

3.5.1. Compiling in a Motorola_68k View or World

After you have developed your Motorola_68k Ada units, you can promote them to installed or to coded using the same approach used in R1000 worlds or views—with [Promote], [Code], [Code (This World)]. When the units in your Motorola_68k view or world have been coded, you will have invoked the assembler automatically (and the linker if the unit is a main procedure) to produce object modules and an executable module. Although it is possible to invoke the compiler alone, the normal operation is to invoke all three together.

3.5.2. Assembling in the Motorola_68k View or World

Although you normally invoke the assembler automatically when you invoke the compiler, you can invoke the assembler explicitly if you have special assembly 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 unit where the Ada specification corresponding to the assembly-language body is declared:

- Interface pragma
- Import_Function pragma
- Import_Procedure pragma

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.

To assemble the units:

1. From a Command window, enter `M68k.Assemble` and press [Complete].
2. Enter the name of the input file that contains the assembly source code at the `Source_File` prompt.
3. Enter the name of the output file that will contain the assembly object code at the `Object_File` prompt.
4. Enter the name of the output file that will contain the assembly listing at the `Listing_File` prompt.
5. Enter `True` at the `Produce_Listing` prompt.
6. Press [Promote].

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 using the assembler, consult Chapter 5, "M68K/OS-9 Cross-Assembler."

3.5.3. Linking in the Motorola_68k View or World

You normally invoke the linker automatically when you invoke the compiler. You can still 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. However, if you have special requirements that necessitate using the linker manually, you can invoke the linker explicitly, but you will be responsible for determining the proper linking order and where in memory the program will reside. To do this, you may have to alter substantially the linker-command file. It is strongly suggested that you invoke the linker automatically and use the standard linker-command file that is provided or use a slightly modified linker-command file. For information on the linker-command file, consult Chapter 6, "M68K/OS-9 Cross-Linker."

If you have produced your own linker-command file, you are ready to invoke the linker manually.

To invoke the linker:

1. From a Command window, enter `M68k.Link` and press [Complete].
2. Enter the name of the file that contains the linker commands at the `Command_File` prompt.
3. Enter the name of the file that will contain the executable object module at the `Exe_File` prompt.
4. Enter `True` in response to the `Produce_Statistics` prompt.
5. Press [Promote].

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 that were linked, the number of symbols that were produced, and the number of fixups that were required and appends it to the link map:

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

3.5.4. Associated Files

When you compile programs, the compiler produces special files called *associated* files, 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 M68K CDF. These files can be specified using 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.

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 Main pragma):

- <Elab_Asm>: Contains the assembly-language source generated by the compiler for the associated 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 run-time 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 M68K CDF is invoked and the Cross_Cg.Listing switch is set to true:

```
= Rational (Delta1) WJH.HEINZE

!Users.Wjh.Demo

  Bench               : Ada (Pack_Spec);
    .<List>           : File;
    .<Obj>            : File;
  Benchmark_Switches  : File (Switch);
  Guidance            : Ada (Pack_Spec);
    .<List>           : File;
    .<Obj>            : File;
  Guidance            : Ada (Pack_Body);
    .<List>           : File;
    .<Obj>            : File;
    .Trajectory       : Ada (Proc_Body);
      .<List>         : File;
      .<Obj>          : File;
    .Range            : Ada (Proc_Body);
      .<List>         : File;
      .<Obj>          : File;
  Main                : Ada (Proc_Spec);
    .<List>           : File;
    .<Obj>            : File;
  Main                : Ada (Main_Body);
    .<Elab_List>      : File;
    .<Elab_Obj>       : File;
    .<Exe>            : File;
    .<Link_Map>       : File;
    .<List>           : File;
    .<Obj>            : File;
  Typedefs            : Ada (Pack_Spec);
    .<List>           : File;
    .<Obj>            : File;
```

3.6. Converting and Transferring Executable Modules

Before you can run executable modules, you must convert them to an object-module format that executes on the D85 hardware and then transfer them to the D85 hardware.

3.6.1. Converting Executable Modules

The executable module produced by the M68K/OS-9 linker is in the R1000 object-module format. To run on the D85 hardware, it must be converted to the OS-9 object-module format.

To convert an executable module:

1. From the library that contains the executable module, create a Command window.

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 D85 hardware at the `New_Module` prompt.

5. Enter `Os9` at the `New_Format` prompt.

6. Press [Promote].

For example, the following command converts the object-module format from Rational to OS-9:

```
Convert (Old_Module => "Main_68k.<exe>",
         New_Module => "Main_68k_Os9",
         New_Format => "Os9");
```

3.6.2. Transferring Executable Modules

The executable module is now in the OS-9 object-module format, so you can transfer it to the D85 hardware.

To transfer an executable module:

1. From the library containing the executable module, create a Command window.

2. Enter `Os9_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 D85 hardware at the `To_Remote_File` prompt. (If you want to debug this program later, it must 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 D85 hardware:

```
Os9_Put (From_Local_File => "Main_Motorola_68k_Os9",
         To_Remote_File => "Main_Motorola_68k_Os9",
         Remote_Machine => Ftp_Profile.Remote_Machine,
         Remote_Directory => Ftp_Profile.Remote_Directory,
         Text_File => False,
         Response => Profile.Get);
```

3.7. Executing and Debugging

Now that you have produced an executable module and have transferred it to the D85 hardware, you are ready to run it on D85 hardware. This is accomplished by running your program directly on the D85 hardware with commands from the OS-9 operating system or by using the R1000-hosted M68K cross-debugger.

To execute the program on target hardware:

1. From a console attached to the D85 hardware, log into the D85 hardware.

2. Go to the directory that contains the OS-9-formatted executable module.

3. Enter the name of the executable module (for example, `Main_M68k_Os9`) and press [Return].

The program now runs on the D85 hardware (however, it does not run under debugger control).

To obtain diagnostic messages, two parameters can be used when entering the program name:

- -d: Specifies task and elaboration diagnostics.

- -s: Specifies storage diagnostics.

The parameters are separated from the program name by at least one space; if both are used, they are separated from each other by at least one space. The following is an example of an execution command with both parameters:

```
Main_M68k_Os9 -d -s
```

To execute the program on target hardware using the M68K cross-debugger:

1. With your library FTP switches set to the remote machine and remote directory that contains the program that you will debug, select the program to be debugged (the unit that contains the Main pragma—that is, the unit whose body has the .<Exe> file associated with it; the program on the D85 hardware must have the same name).

2. From the library containing the selected program, create a Command window.

3. Enter `Debug.Invoke` and press [Complete].

4. Press [Promote].

   The M68K cross-debugger window appears. The debugger is now running and you are ready to execute and debug your programs. Source-level debugging is identical to debugging an R1000 program. Program output will appear on the OS-9 console connected to the D85 hardware.

5. Press [Execute].

To quit the debugger, enter the following command:

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

This command kills the R1000 and OS-9 components of the M68K 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 D85 hardware, see Chapter 9, "M68K/OS-9 Cross-Debugger," and the release note.

4. M68K/OS-9 Cross-Compiler

The M68K/OS-9 Cross-Development Facility provides the user the ability to develop and compile programs using the Rational Environment. However, choosing the Motorola_68k target key selects the M68K code generator instead of the R1000 code generator. The output of the M68K code generator is M68K assembly source code. Under default conditions, the 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 M68K code generators are invisible to the user, and compilation using the M68K/OS-9 Cross-Development Facility is identical to compilation in the Rational Environment.

4.1. Compilation States

To compile a program, a user begins with an object, the Ada unit. An Ada unit can exist in one of three states:

- Source
- Installed
- Coded

It is important to realize that an Ada unit can exist in only one of these states at a given time. Additionally, these states are sequential; a unit cannot go directly from the source state to the coded state without first going through the installed state.

The two compilation models of concern are:

- R1000
- Motorola_68k

4.1.1. Source State

In the source state, an Ada unit contains source code that will be compiled. The Ada unit is registered with the Environment as an anonymous unit; for example, it is represented in the world containing it as _Ada_10. This unit cannot be *with*ed by another unit. The source code 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 model and the Motorola_68k model.

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).

4.1.2. Installed State

The installed state is the 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, the Environment checks to see if the other units are also installed. Although users cannot freely add code, delete code, or edit code in units in the installed state, they can still perform incremental operations (see "Incremental Operations," in this chapter). The installed state is the same in the R1000 model and the Motorola_68k model.

The installed state can be promoted to the next higher state, the installed 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).

4.1.3. Coded State

In the coded state, the following differences exist between the R1000 model and the Motorola_68k model:

- In the R1000 model, the Ada unit can now be executed. If the unit *with*s any other Ada units, the Environment checks to see that the other units are in the coded state; if not, the Environment manages the unit dependencies within the program, ensuring that all units are in the coded state. If the *with*ed units do not exist or cannot be coded, the Environment provides the user with 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).

  When executing an Ada unit, the user must know which unit is the main unit. A Main pragma optionally may be added to the unit to indicate that it is the main unit.

  Figure 4-1 shows the R1000 compilation model.

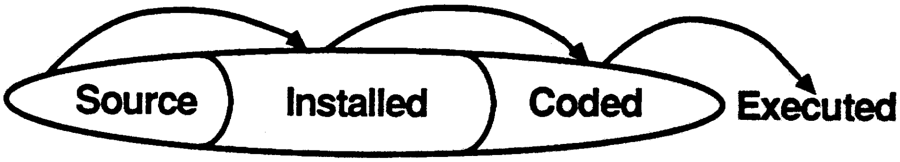

Figure 4-1 R1000 Compilation Model

- In the Motorola_68k model, the Ada unit cannot be executed until some additional steps are taken. By default, the output of the M68K code generator is assembly source for each Ada library unit. The assembly source is assembled by the assembler into relocatable object modules and the assembly source is deleted (see Chapter 5, "M68K/OS-9 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. Although, by default, the code generator, assembler, and linker are invoked automatically to produce the executable module, it is possible to invoke each separately (see the section on library switches in Chapter 3, "Using the M68K/OS-9 Cross-Development Facility").

When the Ada unit is promoted to code, files associated with the Ada unit are produced. These files include the following:

— <Asm>

— <List>

— <Obj>

— <Elab_Asm>

— <Elab_List>

— <Elab_Obj>

— <Exe>

— <Link_Map>

Depending on the switch settings, these files may or may not be retained (see Chapter 3, "Using the M68K/OS-9 Cross-Development Facility," for a discussion of the associated files).

For the linker to know that it needs to produce an executable module for the program, the spec or body of the main Ada unit must contain a Main pragma. 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 the unit is coded, a relocatable object module for that unit is generated along with an elaboration module for the entire program. Then the M68K linker is invoked automatically, and it links all of the program's object modules along with any necessary modules from the Ada run-time library to produce an executable module (see Chapter 6, "M68K/OS-9 Cross-Linker," for more details on the linking process).

Before it can be executed, the executable module must have its object-module format changed, and then it must be downloaded to the D85 hardware (see Chapter 8, "M68K/OS-9 Downloader," for more information on the conversion and downloading processes).

The Ada unit can be demoted from the coded state to a lower state (see "Compiler Commands," below, for a description of commands that can be used to demote an Ada unit from the coded to the installed state).

Figure 4-2 shows the Motorola_68k compilation model.

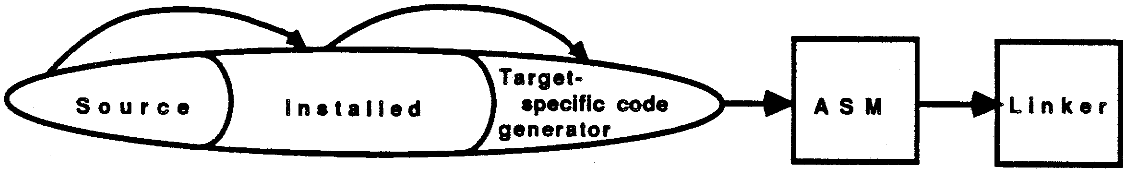

Figure 4-2 Motorola_68k Compilation Model

4.2. Compiler Commands

The same commands are used to invoke the M68K compiler that are used to invoke 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 the compiler commands.

Table 4-1 Compiler Commands

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.

Table 4-1 Compiler Commands (continued)

Command	Function
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.
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 current value in the underlying representation.
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 object and any dependent units.
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 are reversible. This command differs from the Destroy procedure, which permanently deletes and expunges 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.

Table 4-1 Compiler Commands (continued)

Command	Function
Compilation.Destroy	Destroys the named object 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 is applied recursively to the named unit and any other units in the specified scope. A unit is not promoted if it is not a legal Ada unit.

4.3. Differences between R1000 and M68K/OS-9 Compilers

This section briefly highlights the differences between the code generated by the R1000 compiler for execution on the R1000 and the code generated by the M68K/OS-9 Cross-Development Facility for execution on M68K/OS-9 hardware or simulator.

4.3.1. Chapter 13 Support

In general, the M68K/OS-9 compiler provides different support for Chapter 13 of the LRM than does the R1000 compiler. For more details, see Appendix F for each, as well as the release note for this product.

4.3.2. Command Windows

Command windows attached to Motorola_68k 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 M68K/OS-9 code. M68K/OS-9 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-9 format, transferring the converted executable module to the OS-9 system, and executing on the actual target hardware.

4.3.3. Generics

The R1000 architecture enables code-shared generics—multiple instantiations of a generic share the same code. The M68K cross-compiler uses macro expansion to implement instantiations of generics, so multiple instantiations yield multiple copies of the code.

4.3.4. Incremental Operations

In an R1000 world, coded package specifications can be changed incrementally. In a Motorola_68k 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 a Motorola_68k 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 on how to perform incremental operations, consult the *Rational Environment Basic Operations* manual.

Table 4-2 Incremental Operations

Object State	Incremental Operation	R1000	M68K
Installed	Add a statement, declaration, or comment.	X	X
Installed	Change a statement, declaration, or comment.	X	X
Installed	Delete a statement, declaration, or comment.	X	X
Coded	Add a comment.	X	X
Coded	Add a statement or a declaration to a package specification.	X	
Coded	Change a comment.	X	
Coded	Change a statement or a declaration to a package specification.	X	
Coded	Delete a comment.	X	X
Coded	Delete a statement or a declaration to a package specification.	X	

4.3.5. Packed Records and Arrays

In an R1000 world, all records and arrays are always packed automatically. In a Motorola_68k world, the user must explicitly use the Pack pragma or a record representation clause to obtain packing.

4.3.6. Record Layout

The R1000 and M68K/OS-9 compilers lay out record fields differently.

4.3.7. Program Elaboration

The elaboration model is different for the M68K/OS-9. This has some impact on certain debugger operations. See Chapter 9 for more details.

5. M68K/OS-9 Cross-Assembler

5.1. Introduction

The M68K/OS-9 cross-assembler is part of Rational's Cross-Development Facilities. A separate assembler is provided for each specific target computer family, although all assemblers implement the same target-independent directives, conditional facilities, macro facilities, and object format. The assemblers are used to 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. M68K-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 M68K is not required to use this material.

5.2. Assembler Command (M68k.Assemble)

The normal operation of the assembler is automatic. When the library Cross_Cg.Auto_Assemble switch is set to true (the default value), the output of the compiler is assembled automatically. A major reason you would want to use the assembler manually is when you have a separate assembly-language source file that you want to assemble.

The assembler command is:

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

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

  Specifies the output file that will contain the assembly object code.

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

  Specifies the output file that will contain the assembly listing.

- `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 := False;`

  Specifies whether an assembly-listing file is generated. The default is false.

- `Produce_Statistics : Boolean := False;`

  Specifies whether statistics of the assembly process are generated. The statistics will be included at the end of the .<List> 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.

5.2.1. 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 you want 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",
                  Object_File => "User_Example_Object_Code",
                  Listing_File => "User_Example_Listing",
                  Produce_Listing => True);
```

5.3. Assembly-Language Source Code

Input to the assembler comes from a compiler or from a text file created using the text editor, or it is moved to the Rational Environment from another computer system. This text file 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.

5.3.1. Source Statements

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#).

5.3.1.1. 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
```

5.3.1.2. 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.

5.3.1.3. 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. Some operators require more than one operand, in which case the operands must be separated by commas.

5.3.1.4. 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 will be ignored. No restrictions are placed on the characters within a comment.

5.3.1.5. 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)
```

5.3.2. Numeric Literals

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

```
base#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
```

5.3.3. Symbols

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

5.3.3.1. 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
^	Caret

The caret is used to provide symbol uniqueness within macros. It is replaced by a decimal character string that is equal to the number of macro expansions performed when the caret is encountered. This character is not allowed within symbols except within a macro definition.

5.3.3.2. 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 a group of special symbols. Special symbols are used to represent implied entities such as the current value of the location counter. The special symbol is:

.	Value of the current location counter

5.3.3.3. 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

clear_bytes:                    ; address in A0, count in D0.W
        tst.w   d0              ; check the count
        beq.s   $done           ; if zero clear nothing
        subq.w  #1,d0           ; adjust count for DBRA instruction
$loop:  clr.b   (a0)+           ; clear a byte
        dbra    d0,$loop        ; loop for the whole block
$done:  rts                     ; return

        .local                  ; new scope
copy_bytes:                     ; source addr in A0, dest addr in A1
                                ; count in D0.W
        tst.w   d0              ; check the count
        beq.s   $done           ; if zero copy nothing
        subq.w  #1,d0           ; adjust count for DBRA instruction
$loop:  move.b  (a0)+,(a1)+     ; copy a byte
        dbra    d0,$loop        ; loop for the whole block
$done:  rts                     ; return
```

In the above example, note that the symbols $LOOP and $DONE are 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.

5.3.3.4. Symbol Resolution

All instruction mnemonics, register names, directives, macro names, and other permanent symbols are keywords that cannot be redefined by the user. For example:

```
    move    equu    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>, (, #, ...)
```

5.3.4. Expression Evaluation

All expressions are evaluated using 64-bit two's complement arithmetic. 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.

5.3.4.1. Unary Operators

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

5.3.4.2. 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

5.3.4.3. Operator Precedence

Operator precedence is, lowest to highest, as follows:

= /= < > <= >=

! \

&

+ –

* /

** >> <<

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

5.4. Assembler Directives

5.4.1. Listing Directives

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

.LISTNC	List no conditionals
.LISTTC	List true conditionals only (default)
.LISTC	List all conditionals
.LISTMX	List macro expansion (default)
.LISTMC	List macro calls (default)
.LISTNM	List no macro expansions or calls
.LIST	Enable listing
.NLIST	Disable listing (default)
.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

5.4.2. 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.

5.4.2.1. Uninitialized Block Storage

Each of these directives reserves storage for the specified number of elements by advancing the location counter as necessary. These directives are followed by an expression that represents the number of storage units to be allocated. For example:

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

.DS.B Reserve storage for bytes (8 bits)

.DS.W Reserve storage for words (16 bits)

.DS.L Reserve storage for long words (32 bits)

.DS.S Reserve storage for single-precision floating point (target-specific)

.DS.D Reserve storage for double-precision floating point (target-specific)

.DS.X Reserve storage for extended-precision floating point (target-specific)

.DS.A Reserve storage for an address (target-specific)

5.4.2.2. Initialized Unit Allocation

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 long words (32 bits)

.DC.S Define constant single-precision floating point (target-specific)

.DC.D Define constant double-precision floating point (target-specific)

.DC.X Define constant extended-precision floating point (target-specific)

.DC.A Define constant addresses (target-specific)

.ASCII Define a constant string, 8 bits per character

.ASCIZ Define a constant text string, 8 bits per character terminated with a null character

5.4.2.3. Initialized Block Allocation

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 long words (32 bits)
.DCB.S	Define constant block single-precision floating point (target-specific)
.DCB.D	Define constant block double-precision floating point (target-specific)
.DCB.X	Define constant block extended-precision floating point (target-specific)
.DCB.A	Define constant block addresses (target-specific)

5.4.3. Intermodule Symbol-Definition Directive

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)
.EXT.B	Byte external (8 bits)
.EXT.W	Word external (16 bits)
.EXT.L	Long external (32 bits)

5.4.4. Symbol-Definition Directive

The following directives are used to assign symbolic names to expressions. Two forms of directives are available. The first form allows the user to assign a symbol a value and a size attribute. These attributes allow the assembler to generate optimal code even if the value of the symbol cannot yet be determined.

```
.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. Even though it has a 16-bit size, the expression CONTROLLER+3 has an unknown size. The first MOVE instruction will require four bytes to express 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 will 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 M68K family.

The second form of symbol definition directive simply creates a symbol of implied size. An example of the second form is:

```
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.

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
sym1 equ sym2+10
sym2 set 10
```

In this example, SYM1 has a value of 10 throughout the assembly, because at the time it was defined, SYM2 had a value of 0.

.DEFP.B	Define a permanent byte symbol (8 bits)
.DEFP.W	Define a permanent word symbol (16 bits)
.DEFP.L	Define a permanent long word 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 long word 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

5.4.5. Miscellaneous Directives

5.4.5.1. The 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.

5.4.5.2. The SECT Directive

The .SECT directive is used to define, or switch between, program sections. When the .SECT 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 on processor implementations that have physically distinct instruction and data address spaces.
- DATA: The section contains constant data or variable data or both. This attribute is important only on processor implementations 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. A linker option can be used to ensure that the data being written are identical to the data being overwritten.
- 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
```

5.4.5.3. The 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;

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. Any attempts to generate code within the NULL section, with either assembler instructions or directives, is illegal.

5.4.5.4. The 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.

5.4.5.5. The IRADIX Directive

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

5.4.5.6. The ORADIX Directive

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

5.4.5.7. The 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 linkage map. This is useful for tracking module revision level. Compiler-generated code usually outputs the revision of the compiler with this directive.

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

5.4.5.8. The 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 must be a positive power of 2 that is 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 long-word-aligned.

5.4.5.9. The 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
```

5.4.5.10. The 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 will cause the object module produced by the assembler to be marked as containing errors. Linking such modules will produce link-time warnings.

5.4.5.11. The 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 should be a fully qualified pathname to the file. For example:

```
.include    "!users.wjh.project.data_definitions"
```

5.5. Repetitive Assembly and Conditional Assembly

5.5.1. 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 by hand. 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 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 may be indexed by N to get N factorial.

5.5.2. 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:
        .elsif cpu=mc68020                  ;don't emulate if 68020
            cmp2.w  (a0),d0
        .else                               ;don't know this CPU
            .output cpu
            .error  "Unknown processor."
        .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.

5.6. Character Usage

A–Z	User symbol characters
a–z	User symbol characters
0–9	User symbol characters and numeric literals
!	The logical inclusive OR operator
#	Used for immediate operands and based numeric literals
$	User symbol character
%	Used to insert formal parameters into the definition of a macro
&	The logical AND operator
*	The multiplication operator

( ) Used for operator precedence control and effective address syntax

– The unary negation and two’s complement subtraction operator

_ User symbol character (underscore), ignored within numeric literals

+ The 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

" Used with storage directives for creating ASCII strings and to provide string arguments to some directives

\ The logical exclusive OR operator

| The line-continuation character

< Used for relational operators

> Used for relational operators

? User symbol character

, Used as a separator

. User symbol character

~ The one’s complement unary operator

^ Used to pass commas as arguments to macro calls and to generate unique labels within macro body definitions

‘ Not used, illegal

space A separator

ascii.ht A separator

ascii.lf A statement separator

ascii.ff A page delimiter

5.7. Syntax of Assembler Commands

5.7.1. Backus-Naur Formalism (BNF) Used with Assembler Commands

The following BNF is used to define the syntax for assembler 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 any of the following symbols AA or AA,BB or AA,BB,BB or AA,BB,BB,BB and so on are valid.

5.7.2. 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   --> .CPU string              |
                    .LOCAL                   |
                    .PUSH expression         |
                    .RADIX expression        |
                    .IRADIX expression       |
                    .ORADIX expression       |
                    .INCLUDE string          |
                    .REV string              |
                    .ERROR string            |

output          --> .OUTPUT string           |
                    .OUTPUT expression

section         --> .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
                    {.ELSIF 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 '
```

5.7.3. M68K-Family-Dependent 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
```

5.7.4. M68K-Family Instruction Mnemonics

M68K-Family Instruction Mnemonics

Mnemonic	B	W	L	S	D	X	P	Notes
ABCD	X							
ADD	X	X	X					
ADDA		X	X					
ADDI	X	X	X					
ADDQ	X	X	X					
ADDX	X	X	X					
AND	X	X	X					
ANDI	X	X	X					
ANDI to CCR	X							
ANDI to SR		X						
ASL	X	X	X					1
ASR	X	X	X					1
BCC	X	X	X					2
BCHG	X		X					3
BCLR	X		X					3
BCS	X	X	X					2
BEQ	X	X	X					2
BFCHG								5
BFCLR								5
BFEXTS								5
BFEXTU								5
BFFFO								5
BFINS								5
BFSET								5
BFTST								5
BGE	X	X	X					2
BGT	X	X	X					2
BHI	X	X	X					2
BHS	X	X	X					2
BKPT								

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
BLE	X	X	X					2
BLO	X	X	X					2
BLS	X	X	X					2
BLT	X	X	X					2
BMI	X	X	X					2
BNE	X	X	X					2
BPL	X	X	X					2
BRA	X	X	X					2
BSET	X		X					3
BSR	X	X	X					2
BTST	X		X					3
BVC	X	X	X					2
BVS	X	X	X					2
CALLM								
CAS	X	X	X					
CAS2		X	X					
CHK		X	X					
CHK2	X	X	X					
CLR	X	X	X					
CMP	X	X	X					
CMP2	X	X	X					
CMPA		X	X					
CMPI	X	X	X					
CMPM	X	X	X					
DBCC		X						
DBCS		X						
DBEQ		X						
DBF		X						
DBGE		X						
DBGT		X						
DBHI		X						
DBHS		X						

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
DBLE		X						
DBLO		X						
DBLS		X						
DBLT		X						
DBMI		X						
DBNE		X						
DBPL		X						
DBRA		X						
DBT		X						
DBVC		X						
DBVS		X						
DIVS		X	X					4
DIVSL		X	X					5
DIVU		X	X					4
DIVUL		X	X					5
EOR	X	X	X					
EORI	X	X	X					
EORI to CCR	X							
EORI to SR		X						
EXG			X					
EXT		X	X					
EXTB			X					5
ILLEGAL								
JMP		X	X					
JSR		X	X					
LEA			X					
LINK		X	X					4
LSL	X	X	X					1
LSR	X	X	X					1
MOVE	X	X	X					
MOVE to CCR		X						
MOVE to SR		X						

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
MOVE to USP			X					
MOVE from CCR		X						
MOVE from SR		X						
MOVE from USP			X					
MOVEA		X	X					
MOVEC			X					5
MOVEM		X	X					
MOVEP		X	X					
MOVEQ		X	X					
MOVES	X	X	X					6
MULS		X	X					4
MULU		X	X					4
NBCD	X							
NEG	X	X	X					
NEGX	X	X	X					
NOP								
NOT								
OR	X	X	X					
ORI	X	X	X					
ORI to CCR		X						5
ORI to SR		X						5
PACK								5
PEA			X					
RESET								
ROL	X	X	X					1
ROR	X	X	X					1
ROXL	X	X	X					1
ROXR	X	X	X					1
RTD		X						6
RTE								
RTM								5
RTR								

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
RTS								
SBCD	X							
SCC	X							
SCS	X							
SEQ	X							
SF	X							
SGE	X							
SGT	X							
SHI	X							
SHS	X							
SLE	X							
SLO	X							
SLS	X							
SLT	X							
SMI	X							
SNE	X							
SPL	X							
ST	X							
STOP								
SUB	X	X	X					
SUBA		X	X					
SUBI	X	X	X					
SUBQ	X	X	X					
SUBX	X	X	X					
SVC	X							
SVS	X							
SWAP		X						
TAS	X							
TRAP								
TRAPCC		X	X					5,7
TRAPCS		X	X					5,7
TRAPEQ		X	X					5,7

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
TRAPF		X	X					5,7
TRAPGE		X	X					5,7
TRAPGT		X	X					5,7
TRAPHI		X	X					5,7
TRAPHS		X	X					5,7
TRAPLE		X	X					5,7
TRAPLO		X	X					5,7
TRAPLS		X	X					5,7
TRAPLT		X	X					5,7
TRAPMI		X	X					5,7
TRAPNE		X	X					5,7
TRAPPL		X	X					5,7
TRAPT		X	X					5,7
TRAPV								
TRAPVC		X	X					5,7
TRAPVS		X	X					5,7
TST	X	X	X					
UNLK								
UNPK								5
PBAC		X	X					
PBAS		X	X					
PBBC		X	X					
PBBS		X	X					
PBCC		X	X					
PBCS		X	X					
PBGC		X	X					
PBGS		X	X					
PBIC		X	X					
PBIS		X	X					
PBLC		X	X					
PBLS		X	X					
PBSC		X	X					

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
PBSS		X	X					
PBWC		X	X					
PBWS		X	X					
PDBAC		X						
PDBAS		X						
PDBBC		X						
PDBBS		X						
PDBCC		X						
PDBCS		X						
PDBGC		X						
PDBGS		X						
PDBIC		X						
PDBIS		X						
PDBLC		X						
PDBLS		X						
PDBSC		X						
PDBSS		X						
PDBWC		X						
PDBWS		X						
PFLUSH								
PFLUSHA								
PFLUSHR								
PFLUSHS								
PLOADR								
PLOADW								
PMOVE	X	X	X					8
PRESTORE								
PSAVE								
PSAC	X							
PSAS	X							
PSBC	X							
PSBS	X							

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
PSCC	X							
PSCS	X							
PSGC	X							
PSGS	X							
PSIC	X							
PSIS	X							
PSLC	X							
PSLS	X							
PSSC	X							
PSSS	X							
PSWC	X							
PSWS	X							
PTESTR								
PTESTW								
PTRAPAC		X	X					7
PTRAPAS		X	X					7
PTRAPBC		X	X					7
PTRAPBS		X	X					7
PTRAPCC		X	X					7
PTRAPCS		X	X					7
PTRAPGC		X	X					7
PTRAPGS		X	X					7
PTRAPIC		X	X					7
PTRAPIS		X	X					7
PTRAPLC		X	X					7
PTRAPLS		X	X					7
PTRAPSC		X	X					7
PTRAPSS		X	X					7
PTRAPWC		X	X					7
PTRAPWS		X	X					7
PVALID			X					
FABS	X	X	X	X	X	X	X	

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
FACOS	X	X	X	X	X	X	X	
FADD	X	X	X	X	X	X	X	
FASIN	X	X	X	X	X	X	X	
FATAN	X	X	X	X	X	X	X	
FATANH	X	X	X	X	X	X	X	
FBEQ		X	X					
FBF		X	X					
FBGE		X	X					
FBGL		X	X					
FBGLE		X	X					
FBGT		X	X					
FBLE		X	X					
FBNE		X	X					
FBNGE		X	X					
FBNGL		X	X					
FBNGLE		X	X					
FBNGT		X	X					
FBNLE		X	X					
FBNLT		X	X					
FBOGE		X	X					
FBOGL		X	X					
FBOGT		X	X					
FBOLE		X	X					
FBOLT		X	X					
FBON		X	X					
FBOR		X	X					
FBSEQ		X	X					
FBSF		X	X					
FBSNE		X	X					
FBST		X	X					
FBT		X	X					
FBUEQ		X	X					

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
FBUGE		X	X					
FBUGT		X	X					
FBULE		X	X					
FBULT		X	X					
FCMP	X	X	X	X	X	X	X	
FCOS	X	X	X	X	X	X	X	
FCOSH	X	X	X	X	X	X	X	
FDBEQ		X						
FDBF		X						
FDBGE		X						
FDBGL		X						
FDBGLE		X						
FDBGT		X						
FDBLE		X						
FDBLT		X						
FDBNE		X						
FDBNGE		X						
FDBNGL		X						
FDBNGLE		X						
FDBNGT		X						
FDBNLE		X						
FDBNLT		X						
FDBOGE		X						
FDBOGL		X						
FDBOGT		X						
FDBOLE		X						
FDBOLT		X						
FDBON		X						
FDBOR		X						
FDBSEQ		X						
FDBSF		X						
FDBUEQ		X						

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
FDBUGE		X						
FDBUGT		X						
FDBULE		X						
FDBULT		X						
FDBSNE		X						
FDBST		X						
FDIV	X	X	X	X	X	X	X	
FETOX	X	X	X	X	X	X	X	
FETOXM1	X	X	X	X	X	X	X	
FGETEXP	X	X	X	X	X	X	X	
FGETMAN	X	X	X	X	X	X	X	
FINT	X	X	X	X	X	X	X	
FINTRZ	X	X	X	X	X	X	X	
FLOG10	X	X	X	X	X	X	X	
FLOG2	X	X	X	X	X	X	X	
FLOGN	X	X	X	X	X	X	X	
FLOGNP1	X	X	X	X	X	X	X	
FMOD	X	X	X	X	X	X	X	
FMOVE	X	X	X	X	X	X	X	
FMOVECR	X	X	X	X	X	X	X	
FMOVEM			X			X		9
FMUL	X	X	X	X	X	X	X	
FNEG	X	X	X	X	X	X	X	
FNOP								
FREM	X	X	X	X	X	X	X	
FRESTORE								
FSAVE								
FSCALE	X	X	X	X	X	X	X	
FSEQ	X							
FSF	X							
FSGE	X							
FSGL	X							

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
FSGLDIV	X	X	X	X	X	X	X	
FSGLE	X							
FSGLMUL	X	X	X	X	X	X	X	
FSGT	X							
FSIN	X	X	X	X	X	X	X	
FSINCOS	X	X	X	X	X	X	X	
FSINH	X	X	X	X	X	X	X	
FSLE	X							
FSLT	X							
FSNE	X							
FSNGE	X							
FSNGL	X							
FSNGLE	X							
FSNGT	X							
FSNLE	X							
FSNLT	X							
FSOGE	X							
FSOGL	X							
FSOGT	X							
FSOLE	X							
FSOLT	X							
FSON	X							
FSOR	X							
FSQRT	X	X	X	X	X	X	X	
FSSEQ	X							
FSSF	X							
FSSNE	X							
FSST	X							
FST	X							
FSUB	X	X	X	X	X	X	X	
FSUEQ	X							
FSUGE	X							

M68K-Family Instruction Mnemonics (continued)								
Mnemonic	**B**	**W**	**L**	**S**	**D**	**X**	**P**	**Notes**
FSUGT	X							
FSULE	X							
FSULT	X							
FTAN	X	X	X	X	X	X	X	
FTANH	X	X	X	X	X	X	X	
FTENOX	X	X	X	X	X	X	X	
FTRAPEQ		X	X					7
FTRAPF		X	X					7
FTRAPGE		X	X					7
FTRAPGL		X	X					7
FTRAPGLE		X	X					7
FTRAPGT		X	X					7
FTRAPLE		X	X					7
FTRAPLT		X	X					7
FTRAPNE		X	X					7
FTRAPNGE		X	X					7
FTRAPNGL		X	X					7
FTRAPNGT		X	X					7
FTRAPNLE		X	X					7
FTRAPNLT		X	X					7
FTRAPOGE		X	X					7
FTRAPOGL		X	X					7
FTRAP0GT		X	X					7
FTRAPOLE		X	X					7
FTRAPOLT		X	X					7
FTRAPON		X	X					7
FTRAPOR		X	X					7
FTRAPSEQ		X	X					7
FTRAPSF		X	X					7
FTRAPSNE		X	X					7
FTRAPST		X	X					7
FTRAPUEQ		X	X					7

M68K-Family Instruction Mnemonics (continued)

Mnemonic	B	W	L	S	D	X	P	Notes
FTRAPUGE		X	X					7
FTRAPUGT		X	X					7
FTRAPULE		X	X					7
FTRAPULT		X	X					7
FTRAPT		X	X					7
FTST	X	X	X	X	X	X	X	
FTWOTOX	X	X	X	X	X	X	X	

1. These shift and rotate type instructions allow only WORD operands if the destination is a memory location. BYTE, WORD, and LONG operands are allowed when the operation is performed on a data register.

2. These PC-relative branch instructions support LONG displacements only on the MC68020 implementation. MC68000 and MC68010 support only BYTE and WORD displacements.

3. These bit instructions allow only BYTE operands if the bit is within a memory location. LONG operands are allowed only if the operation is performed on a data register.

4. Only the MC68020 implementation supports the LONG form of this instruction. All processors support the WORD form.

5. This instruction is supported only by the MC68020 implementation.

6. This instruction is supported by the MC68010 and MC68020 implementations but not the MC68000.

7. These instructions take no operand when used without a size designator.

8. The operand size of the PMOVE instruction is destination-dependent. CRP, SRP, and DPR are DOUBLE LONG operands. TC is a LONG operand. BACn, BADn, AC, PSR, and PCSR are WORD operands. CAL, VAL, and SCC are BYTE operands. The assembler requires that the size designator of the instruction match that of the MMU operand.

9. The FMOVEM instruction allows only EXTENDED operands when moving to or from floating-point registers and only LONG operands when moving to or from floating-point control registers.

6. M68K/OS-9 Cross-Linker

The M68K/OS-9 cross-linker is a part of Rational's Cross-Development Facilities. The same linker is provided for each target computer; however, each target requires a different standard linker-command file. The linker is used to produce executable programs by combining the contents of various object modules, which can be supplied directly to the linker or retrieved from 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. For example, 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.

6.1. 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, generic instantiation, subunit, or task body.

- 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 that is 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.

6.2. Linker Command (M68k.Link)

The normal operation of the linker is automatic. When the Auto_Link library switch is set to true (the default value), the output of the compilation system is linked automatically with other object files (for example, run-time files) to create an executable module when a main program is promoted to coded.

The linker command is:

```
M68k.Link (Command_File : String := "<IMAGE>";
           Exe_File : String := "<DEFAULT>";
           Map_File : String := "<DEFAULT>";
           Debug_Symbol_File : String := "<DEFAULT>";
           Symbol_Table_File : String := "<DEFAULT>";
           Produce_Debug_Table : Boolean := False;
           Produce_Symbol_Table : 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.

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

  Specifies the file that will contain the link map.

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

  Specifies the output file that will contain the debug-symbol table, if one is generated.

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

  Specifies the output file that will contain the symbol table, if one is generated.

- `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_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 run-time 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);
```

6.3. The Linking Process

The following sections discuss the 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 be 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.

6.3.1. 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.

6.3.2. 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 errors later.

6.3.3. 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.

6.3.4. 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 number of segments this program should have, the collections that should be placed in each segment, and the attributes of each segment (for example, read-only records). In addition, the generation of output for a specific segment may be suppressed altogether. This feature may be useful for suppressing the data associated with uninitialized operand spaces in some computers. Every collection defined previously 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 and/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.

6.3.5. 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, 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
- If requested by the link command, a statistical summary of the number of object modules that were linked, the number of symbols that were processed, and the number of fixups required

6.4. Linker-Command Files

The linker reads a textual 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. The Cross-Development Facility was designed so that most users can 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 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 following is an example of a linker-command file:

```
program "Standard_Linker_Commands" is

    link "module_header_asm_object";

    use library "ada_runtime_library";
    collection shared_header    is (module$header);
    collection code             is (ada$runtime, tlcode);
    collection constant_data    is (ada$runtime_const, tlconst);
    collection shared_trailer   is (module$crc);

    collection unshared_header  is (module$writeable_data_begin);
    collection writeable_data   is (ada$runtime_data, tldata);
    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 shared_trailer;
    end;

    segment unshared is

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

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

The symbols can contain from 1 to 32 characters, as specified below.

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
place
program
resolve
section
segment
start
suppress
to
type
use

6.4.1. Basic Commands Used with Linker-Command Files

Each linker command is described in detail in this section. The 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.

6.4.1.1. 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;
```

6.4.1.2. 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";
```

6.4.1.3. 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. It is a string.

Examples of this command are:

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

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

6.4.1.4. 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 to 32 char- acters 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 to 32 characters long.

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) ;
```

6.4.1.5. 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*, followed by a semicolon.

The format of this command is:

> `segment` *segment_id* `is` *{placement} [memory bounds] [segment type]*
> *[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 to 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.

6.4.1.6. 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 to 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#0150# ;
```

6.4.1.7. Memory Bounds

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

The format of this subcommand is:

```
memory bounds are 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#) ;
```

6.4.1.8. Segment Type

The Segment Type command is a subcommand of the linker commands. 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 is used to identify the type of collections found in the segment. It can be 1 to 32 characters long (see above for valid characters).

An example of this subcommand is:

```
segment type is linker_example ;
```

6.4.1.9. Suppress Segment

The Suppress Segment command is a subcommand of the linker commands. It specifies that the segment is to be suppressed. This subcommand is terminated by a semicolon.

The format of this subcommand is:

```
suppress ;
```

An example of this subcommand is:

```
suppress ;
```

6.4.1.10. 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 to 32 characters long (see above for valid characters).

An example of this command is:

```
    exclude section section_name ;
```

6.4.1.11. 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:

```
    resolve | force symbol_name 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 to 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 ;
```

6.4.1.12. 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# ;
```

6.5. Backus-Naur Formalism (BNF) Used with Linker-Command Files

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

- 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.

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_collection }
                        specify_segments { specify_segment }
                        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
```

7. Run-Time Organization

7.1. Introduction

This chapter describes the method by which the M68K/OS-9 Cross-Development Facility translates features of the Ada language onto the instruction-set architecture of the M68K family and the facilities of the OS-9 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.

The user should have knowledge of the Ada language, the M68K-family instruction set, the OS-9 operating system, and the techniques for mapping high-level languages onto computer architectures and operating systems.

7.2. Program Execution Model

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

7.2.1. 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. The program sections currently are named TLCODE, TLCONST, and TLDATA, 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     TLCODE,RELOCATABLE,CODE,READONLY,...
     .SECT     TLCONST,RELOCATABLE,DATA,READONLY,...
     .SECT     TLDATA,RELOCATABLE,DATA,READWRITE,...
```

The linker-command file used for linking a main program specifies the placement of the program sections that constitute the program.

7.2.2. 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.

7.2.3. Processor Resource Utilization

This section discusses how the M68K/OS-9 run-time uses registers and manages memory.

7.2.3.1. Registers

The conventions observed by the Ada run-time model for the usage of the M68K-family registers are described below. Assembly or other language 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 can be used reliably.

- A6 is the frame pointer. The structure of frames built is described later in this document.

- A5 is the global data pointer. Because OS-9 requires that data storage be position-independent, references to statically allocated data must be made indirect through a register. The Ada run-time model uses A5 for data indirection. (Note that many other OS-9 programs, such as those produced by the OS-9 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 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.

7.2.3.2. Memory-Management Options

The run-time 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 run-time system references memory-management hardware.

7.3. 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.

To better illustrate the structure of a frame and the usage of the information contained in a frame, portions of the generated code for the following program fragment will be analyzed:

```
declare

    Global_Variable : Integer := 12;

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

    function Compute_Result
                (Left : Integer; Right : Integer)
            return Integer is
    begin
        return Left + Right;
    end Compute_Result;

begin
    Do_Something (Left => 9, Right => 3);
    Global_Variable := Compute (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.

7.3.1. Stack Structure

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

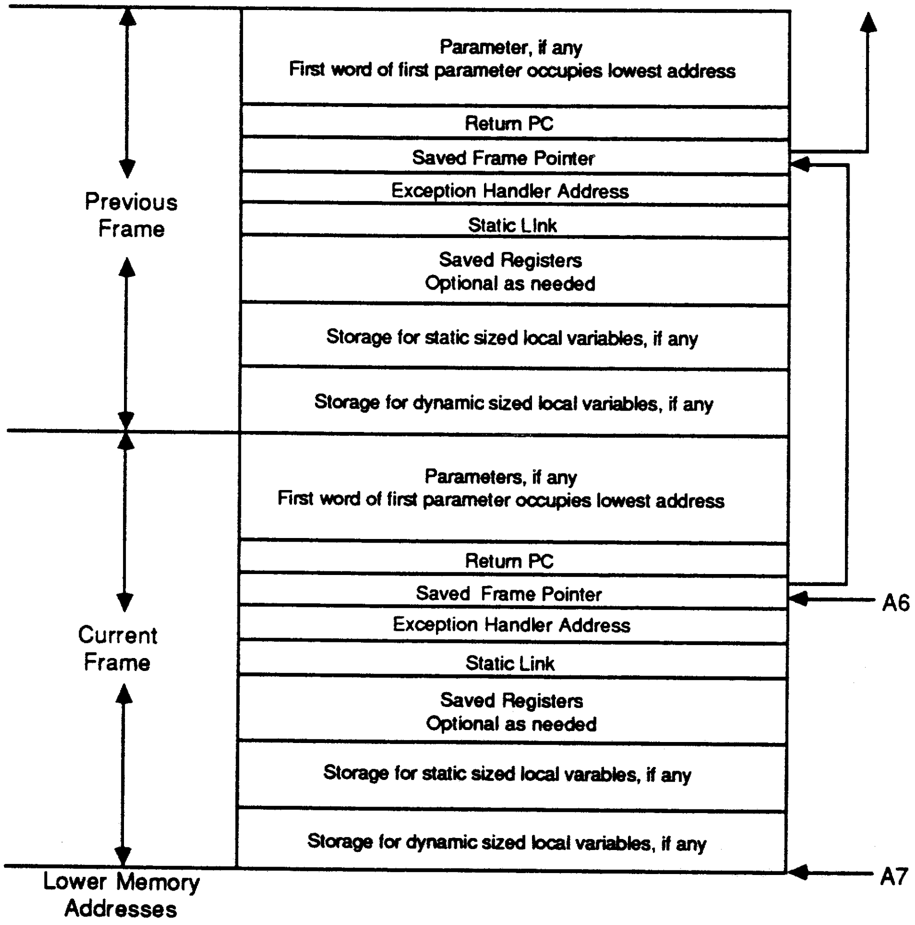
Parameter, if any
First word of first parameter occupies lowest address
Return PC
Saved Frame Pointer
Exception Handler Address
Static Link
Saved Registers
Optional as needed
Storage for static sized local variables, if any
Storage for dynamic sized local variables, if any
Parameters, if any
First word of first parameter occupies lowest address
Return PC
Saved Frame Pointer
Exception Handler Address
Static Link
Saved Registers
Optional as needed
Storage for static sized local varables, if any
Storage for dynamic sized local variables, if any
Previous
Frame
Current
Frame
Lower Memory
Addresses
A6
A7

Figure 7-1 Stack Model

7.3.2. 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                 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 the two words from the stack that were allocated to pass the actual parameters.

The generated code for the body of Do_Something is:

```
Do_Something:
    LINK        A6,#-8                1
    MOVE.L      A1,(-8,A6)            2
    LEA         (Epilog,PC),A0
    MOVE.L      A0,(-4,A6)            3
    MOVE.L      (8,A6),D0             4
    ADD.L       (12,A6),D0            5
    ADD.L       D0,([-8,A6],-8)       6
Epilog:
    LEA         (-8,A6),A7            7
    UNLK        A6                    8
    RTS                               9
```

These instructions perform the following operations:

1. The LINK instruction pushes the frame pointer A6 on the stack, loads A6 with the address of the saved frame pointer, and allocates two more words on the stack to be used for the exception-handler address and for the saved static link. If the subprogram had required additional space for static-sized local variables, that space also would be allocated using the link instruction.

2. The first MOVE.L instruction saves the static link in the frame.

3. The pair of instructions LEA, MOVE.L computes the address of the epilog code (or exception handler, if any) and saves the address in the frame. If this subprogram used any nonvolatile registers, these would be pushed on the stack at this point.

4. The next MOVE.L instruction loads D0 with values of the Right formal parameter.

5. The first ADD.L instruction computes the subexpression Left + Right.

6. The next ADD.L instruction adds the previously computed value to the value of Global_Variable and stores the result in Global_Variable. Note that the addressing of Global_Variable is done using the saved static link.

7. The LEA instruction in the epilog pops the stack back to include only the space allocated for saved registers, local variables, and the objects in the frame. If nonvolatile registers had been saved on entry to the subprogram, they would be restored here.

8. The UNLK instruction pops the stack down to the frame pointer and then pops the value of the saved frame pointer into the frame pointer A6. The return program counter is left as the word on top of the stack.

9. The RTS instruction pops an address from the top of the stack and branches to the instruction at that address.

7.3.3. A Simple Function Call

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

```
    Global_Variable := Compute (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 Global_Variable.

The generated code for the body of the function is:

```
Compute:
    LINK        A6,#-8
    MOVE.L      A1,(-8,A6)
    LEA         (Epilog,PC),A0
    MOVE.L      A0,(-4,A6)
    MOVE.L      (8,A6),D0
    ADD.L       (12,A6),D0
Epilog:
    LEA         (-16,A6),A7
    UNLK        A6
    RTS
```

The body of the function is very similar to that of the procedure analyzed above, except that the result computed by the function is returned in D0.

7.3.4. 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.

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

7.3.4.1. Scalar Types and Access Types

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

7.3.4.2. Simple Record and Array Types

Simple record and array types include nondiscriminated record types, constrained subtypes of discriminated record types, and constrained array subtypes. Parameters of these simple types are passed by reference via a single long word on the stack that contains 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.

7.3.4.3. Discriminated Records of Unconstrained Types

Objects passed as actuals corresponding to formal parameters of unconstrained discriminated record types occupy two long words on the stack. The long word at the higher memory address contains a 0 if the actual is an unconstrained object and a 1 if the actual is a constrained object. The long word at the lower memory address contains the address of the object.

7.3.4.4. Unconstrained Array Types

Objects passed as actuals corresponding to formal parameters of unconstrained types occupy two long words on the stack. The long word at the higher memory address contains the address of a compiler-generated descriptor or dope vector for the array. The long word 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.

7.3.4.5. 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

7.3.4.6. Functions Returning Fixed-Sized Structures

Functions that return fixed-size structures 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.

7.3.4.7. Functions Returning Dynamic-Sized Structures

Several different mechanisms exist for returning dynamic-sized 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.

7.3.5. 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.

7.4. 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, the run times may change the return program counter of the frame to point back into the run times. The run-time exception processor then jumps to the exception-handling code pointed to by the exception-handler address 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 code cuts back the stack, restores all saved registers, and returns. Since the return program counter has been modified, control returns not to the caller of the subprogram but to the run times. The run-time exception processor repeats this process until a frame with exception-handling code is found. The code generated for an exception handler begins with a NOP instruction, which allows the run-time exception processor to distinguish exception handlers from finalization code. The ID of the exception is available to the generated code for the handler in D0.

Exception IDs are currently 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 run-time 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 run-time system when an exception condition is detected

7.4.1. 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

7.4.2. Exceptions Raised by the Run-Time System

The following list indicates the situations in which exceptions are raised by the run-time 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

7.5. Storage Management

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

7.5.1. The Heap

The term *heap* refers to the memory from which collections, task-control blocks, and other run-time 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 run-time 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, Storage_Error is raised.

7.5.2. Collections

A collection is a data structure used by the run-time system to reserve a block of memory for allocation of objects of a given access type. A 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 Storage-_Error.

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 run-time 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.

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

7.5.2.1. 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 suppressed. Note that if the collection is needed, it is created with a default size determined by the run-time system. This storage then is inaccessible to the program throughout its execution.

7.5.2.2. 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.

7.5.3. 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 word size. If the collection's free list does not contain a chunk of memory large enough and the collection is non-extensible 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.

7.5.4. 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.

7.6. Tasking

This section discusses the problems of tasking and the run-time system.

7.6.1. Tasks

An Ada main program runs as a process in OS-9. Additionally, every task object in an Ada program is implemented as a separate OS-9 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 run-time 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 run-time system, as well as one message queue for the main program.

Storage_Error 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.

7.6.2. 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 provided to the run-time system by generated code and can be set to a nondefault value by an argument to the Main pragma. At program initiation, the run-time system queries the OS-9 priority at which the program is executing. These two values determine a correspondence of Ada task priority to OS-9 process priority, which is maintained when a task is created.

For example, if a main program has priority 10 and is run at OS-9 priority 100, a task within the program that has a specified priority of 20 will execute at priority 110 as an OS-9 process.

7.6.3. 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.

7.7. Compiler/Run-Time System Interfaces

A large number of interfaces are used by the compiler to access the run-time system. These are documented below for informational purposes only; run-time interfaces are subject to change with major releases of the compiler.

7.7.1. Attributes

- Enumeration_Image

```
procedure Enumeration_Image (Table  : System.Address;
                             Value  : Integer;
                             String : System.Address;
                             Dope   : System.Address);
```

Called to produce the string for T'Image (X), where T is an enumeration type and X is not static.

- Enumeration_Pos

```
function Enumeration_Pos (Table : System.Address;
                          Value : Integer) return Integer;
```

Called to evaluate T'Pos (X), where T is an enumeration type with a representation specification and X is not static.

- Enumeration_Pred

```
function Enumeration_Pred (Table : System.Address;
                           Value : Integer) return Integer;
```

Called to evaluate T'Pred (X), where T is an enumeration type with a representation specification and X is not static.

- Enumeration_Succ

```
function Enumeration_Succ (Table : System.Address;
                           Value : Integer) return Integer;
```

Called to evaluate T'Pred (X), where T is an enumeration type with a representation specification and X is not static.

- Enumeration_Value

```
function Enumeration_Value (Table  : System.Address;
                            Length : Integer;
                            String : System.Address)
                                              return Integer;
```

Called to evaluate T'Value (X), where T is an enumeration type and X is not static.

- Enumeration_Width

```
function Enumeration_Width (Table : System.Address;
                            Lower : Integer;
                            Upper : Integer) return Integer;
```

Called to evaluate T'Width, where T is a constrained subtype of an enumeration type with dynamic bounds.

- Integer_Image

```
procedure Integer_Image (Value  : Integer;
                         String : System.Address;
                         Dope   : System.Address);
```

Called to evaluate T'Image (X), where T is an integer type and X is not static.

- Integer_Value

```
function Integer_Value (Length : Integer;
                        Value : System.Address) return Integer;
```

Called to evaluate T'Value (X), where T is an integer type and X is not static.

- Integer_Width

```
function Integer_Width (Lower : Integer;
                        Upper : Integer) return Integer;
```

Called to evaluate T'Width, where T is a constrained integer type with dynamic bounds.

7.7.2. Delays

- Delay_Statement

```
procedure Delay_Statement (Delay : Duration;
                           Scale : Integer);
```

Called to evaluate a delay statement.

7.7.3. Exceptions

- Raise_Exception

```
procedure Raise_Exception (Exception_Id : System.Address);
```

Called to raise a specified exception.

- Propagate_Exception

```
procedure Propagate_Exception;
```

Called from an exception handler that contains no alternative for the exception at hand or from finalization code.

- Reraise_Exception

```
procedure Reraise_Exception (Exception_Id : System.Address);
```

Called to evaluate an Ada raise statement that specifies no exception.

- Raise_Constraint_Error

```
procedure Raise_Constraint_Error;
```

- Raise_Numeric_Error

```
procedure Raise_Numeric_Error;
```

- Raise_Program_Error

```
procedure Raise_Program_Error;
```

- Raise_Storage_Error

```
procedure Raise_Storage_Error;
```

- Raise_Tasking_Error

```
procedure Raise_Tasking_Error;
```

7.7.4. Storage Management

- Allocate_Collection

```
function Allocate_Collection (Size       : Integer;
                              Extensible : Boolean)
                                           return System.Address;
```

Called at the point of elaboration of an access-type declaration.

- Allocate_Fixed_Cell

```
function Allocate_Fixed_Cell (Size       : Integer;
                              Collection : System.Address)
                                           return System.Address;
```

Called to evaluate an allocator.

- Collection_Size

```
function Collection_Size (Collection : System.Address)
                                                return Integer;
```

Called to evaluate T'Storage_Size, where T is an access type.

- Deallocate_Collection

```
procedure Deallocate_Collection
                             (Collection : in out System.Address);
```

Called when leaving a block that declared an access type.

- Deallocate_Fixed_Cell

```
procedure Deallocate_Fixed_Cell
                             (Size       : Integer;
                              Collection : System.Address;
                              Cell       : in out System.Address);
```

Called from Unchecked_Deallocation.

7.7.5. Tasking

- Abort_Multiple_Tasks

```
procedure Abort_Multiple_Tasks
                   (Count : Integer;
                    Tasks : array (1..Count) of System.Address);
```

Called for an Ada abort statement that names multiple tasks.

- Abort_Task

```
procedure Abort_Task (Task : System.Address);
```

Called to evaluate an Ada abort statement that names a single task.

- Activate_Offspring

```
procedure Activate_Offspring (Activation_Group : System.Address);
```

Called when entering a block if that block declared objects of task types; called as part of the evaluation of a statement that contains an allocator of task types.

- Await_Dependents

```
procedure Await_Dependents;
```

Called when leaving a block that is the master of one or more tasks.

- Begin_Accept

```
function Begin_Accept (Entry      : Integer;
                       Parameters : in out System.Address)
                                           return System.Address;
```

Called by a task that is entering the synchronization point for accepting a rendezvous.

- Check_Return_Task

```
function Check_Return_Task (Task  : System.Address;
                            Frame : System.Address)
                                           return Boolean;
```

Called to ensure that the task object that is being returned by a function will be in the activation of some block after the function has returned.

- Close_Alternatives

```
procedure Close_Alternatives;
```

Called to close all delay, terminate, and entry alternatives of a task.

- Conditional_Entry_Call

```
function Conditional_Entry_Call
                    (Task       : System.Address;
                     Entry      : Integer;
                     Parameters : System.Address) return Boolean;
```

Called to make a conditional entry call.

- Create_Task

```
function Create_task (Activation_Group : in out System.Address;
                      Master           : System.Address;
                      Starting_Pc      : System.Address;
                      Entry_Count      : Integer;
                      Stack_Size       : Integer;
                      Priority         : Integer;
                      Closure          : System.Address)
                                           return System.Address;
```

Called at the point of elaboration of the declaration of a task object or at the point of evaluation of an allocator of a task type.

- End_Accept

```
procedure End_Accept;
```

Called at the conclusion of the synchronization point for an accept statement.

- Entry_Call

```
procedure Entry_Call (Task       : System.Address;
                      Entry      : Integer;
                      Parameters : System.Address);
```

Called to effect an unconditional, untimed entry call.

- Entry_Count

```
function Entry_Count (Entry : Integer) return Integer;
```

Called to evaluate X'Count, where X names a task entry.

- Initialize_Master

```
function Initialize_Master (Layer : System.Address)
                                      return System.Address;
```

Called as part of the elaboration of a block that is a task master.

- Notify_Parent

```
procedure Notify_Parent (Cause_Tasking_Error : Boolean);
```

Called by a task when it has completed its activation.

- Open_Delay_Alternative

```
procedure Open_Delay_Alternative
                            (Delay          : Duration
                             Scale          : Integer;
                             System.Address : System.Address);
```

Called to indicate that a delay alternative is open for the select statement being evaluated.

- Open_Entry

```
procedure Open_Entry (Entry          : Integer;
                      System.Address : System.Address);
```

Called to indicate that an entry is open for the select statement being evaluated.

- Open_Terminate_Alternative

```
procedure Open_Terminate_Alternative;
```

Called to indicate that a terminate alternative is open for the select statement being evaluated.

- Select_Rendezvous

```
function Select_Rendezvous (Has_Else_Part : Boolean;
                            Parameters    : System.Address)
                                                 return Integer;
```

Called to complete the evaluation of a select statement.

- Synchronization_Point

```
procedure Synchronization_Point;
```

- Task_End

```
procedure Task_End;
```

Called to indicate that the currently executing task is terminating.

- Task_Stack_Size

```
function Task_Stack_Size (Task : System.Address) return Integer;
```

Called to evaluate T'Storage_Size, where T is a task type or an object of a task type.

- Task_Terminated

```
function Task_Terminated (Task : System.Address) return Boolean;
```

Called to evaluate T'Terminated, where T denotes an object of a task type.

- Terminate_Allocated_Offspring

```
procedure Terminate_Allocated_Offspring
                     (Activation_Group : in out System.Address);
```

Called to cause the termination of the offspring of a block that has allocated tasks.

- Terminate_Dependent_Offspring

```
procedure Terminate_Dependent_Offspring;
```

Called to terminate all offspring tasks of the current task.

- Timed_Entry_Call

```
function Timed_Entry_Call (Task       : System.Address;
                           Entry      : Integer;
                           Delay      : Duration;
                           Scale      : Integer;
                           Parameters : System.Address)
                                                return Boolean;
```

Called to evaluate a timed entry call.

7.7.6. Utilities

- Stack_Check

```
procedure Stack_Check (Space : Integer);
```

Called whenever stack space is about to be allocated to ensure that the desired amount of storage is available.

7.7.7. Miscellaneous

- Start_Tasking

```
procedure Start_tasking (Return_Pc : System.Address);
```

Called to initialize the run-time system for a tasking program.

- Middle_Tasking

```
    procedure Middle_Tasking;
```

  Called to indicate to the run-time system that a tasking program has finished its elaboration and is about to begin execution.

- Finish_Tasking

```
    procedure End_Tasking;
```

  Called to indicate to the run-time system that the task that is the main program has terminated.

- Start_Sequential

```
    procedure Start_Sequential (Return_Pc : System.Address);
```

  Called to initialize the run-time system for a sequential program.

- Middle_Sequential

```
    procedure Middle_Sequential;
```

  Called to indicate to the run-time system that a sequential program has finished its elaboration and is about to begin execution.

- Finish_Sequential

```
    procedure Finish_Sequential;
```

  Called to indicate to the run-time system that the main program has terminated.

8. M68K/OS-9 Downloader

The M68K/OS-9 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-9 format. The M68K/OS-9 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 D85 hardware. The M68K/OS-9 CDF provides the Os9_Put command to download the executable module. Once on the D85 hardware, the file can be executed directly using the OS-9 operating-system commands. Alternatively, it can be executed through the use of the M68K/OS-9 cross-debugger (see Chapter 9).

8.1. Format-Conversion Command (Convert)

The output of the M68K/OS-9 linker is an executable module in the R1000 object-module format. However, this format will not execute on the D85 hardware. The format must first be converted to the OS-9 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);
```

The parameters of this command are:

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

  Specifies the name of the executable module that contains a non-OS-9-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;`

  Specifies the object-module format of the new module.

8.2. Converting the Executable Files

The executable file produced by the M68K/OS-9 linker is in the R1000 object-module format. To run on the D85 hardware, it must be converted to the OS-9 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 D85 hardware at the `New_Module` prompt.

5. Enter `Os9` at the `New_Format` prompt.

6. Press [Promote].

For example, the following command converts the object-module format from Rational to OS-9:

```
Convert (Old_Module => "Main_68k.<exe>",
         Old_Format => "Rational",
         New_Module => "Main_68k_Os9",
         New_Format => "Os9");
```

8.3. Transfer Command (Os9_Put)

You must now transfer the executable module that has the OS-9 object-module format to the D85 hardware.

The transfer command is:

```
Os9_Put (From_Local_File : String := "<IMAGE>";
         To_Remote_File : String := "";
         Remote_Machine : String := Ftp_Profile.Remote_Machine;
         Remote_Directory : String :=
                                  Ftp_Profile.Remote_Directory;
         Text_File : Boolean := False;
         Response : Profile.Response_Profile := Profile.Get);
```

The parameters for this command are:

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

  Specifies the name of the file on the R1000 that contains the OS-9-compatible executable module.

- `To_Remote_File : String := "";`

  Specifies the name of the executable module on the D85 hardware. The default indicates that the name of the file on the R1000 will be used. If you do not use the same filename as on the R1000, you will not be able to debug your file.

- `Remote_Machine : String := Ftp_Profile.Remote_Machine;`

  Specifies the name of the remote machine to which the executable module is transferred. (If you have set the Ftp_Profile.Remote_Machine switch in your switch file, you can use the default.)

- `Remote_Directory : String := Ftp_Profile.Remote_Directory;`

  Specifies the name of the remote directory that will receive the transferred executable module. (If you have set the Ftp_Profile.Remote_Directory switch in your switch file, you can use the default.)

- `Text_File : 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.Get;`

  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.

8.4. Transferring the Executable Files

The executable module is now in the OS-9 object-module format, and you can transfer it to the D85 hardware. To accomplish this, perform the following steps:

1. Create a Command window off the library containing the executable module.

2. Enter `Os9_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 D85 hardware 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_Os9 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):

```
Os9_Put (From_Local_File => "Main_68k_Os9",
         To_Remote_File => "Main_68k_Os9",
         Remote_Machine : String := Ftp_Profile.Remote_Machine;
         Remote_Directory : String :=
                                    Ftp_Profile.Remote_Directory;
         Text_File : Boolean := False;
         Response : Profile.Response_Profile := Profile.Get);
```

8.5. Command Used to Execute Directly on the D85 Hardware

To run your executable module on the D85 hardware, you enter the filename of the module from a console connected to the D85 hardware. However, you must be in the directory on the remote machine that contains the executable module.

The command is:

executable_module_name `-d -s`

The parameters are:

- *executable_module_name*: Specifies the filename of the executable module that you transferred to the D85 hardware.
- `-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.
- `-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_os9 -d -s
```

executes the executable module found in Main_68k_Os9 and generates elaboration, task, and storage diagnostics.

9. M68K Cross-Debugger

The M68K/OS-9 Cross-Development Facility provides the user with the ability to debug programs running on D85 hardware. Choosing the Motorola_68k target key selects the M68K cross-debugger instead of the R1000 debugger. The same interface (!Commands-.Debug) is used to control both the R1000 debugger and the M68K cross-debugger.

A given session may be running multiple debuggers of different target types. When commands are entered into a new Debugger window, the new debugger becomes the current debugger. The Debug.Current_Debugger command can also designate one of these debuggers as the current debugger. All subsequent commands are directed to that debugger.

9.1. Commands Used with the M68K Cross-Debugger

For a full discussion on using the debugger, consult the Debugging (DEB) book of the *Rational Environment Reference Manual*.

Table 9-1 lists the commands that are used with the M68K cross-debugger.

Table 9-1 Debug Commands

Debugging Command	Function
Debug.Address_To_Location	Displays the Ada source-code location of the specified address.
Debug.Current_Debugger	Establishes an M68K cross-debugger as the current debugger.
Debug.Invoke	Starts the debugger on the selected main unit after determining the target key.
Debug.Kill	Terminates an M68K debugging session.
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 a word of memory.
Debug.Object_Location	Displays the machine address of the specified object (variable).
Debug.Register_Display	Displays the 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 machine instruction level.
M68k_Debugger	Starts debugging of the selected main unit.

9.1.1. Invoking the Debugger

There is no accelerated key binding for invoking an M68K cross-debugger. To start the M68K cross-debugger on a main program, select the unit and then execute one of the following commands:

- Debug.Invoke
- M68k_Debugger

9.1.1.1. Debug.Invoke

This command starts the debugger on the selected main unit after determining the target key. If a previous M68K debugger still exists, that debugger is used rather than a newly created one. Optionally, the name of the remote machine and the directory on the remote machine that contains the transferred executable module in the OS-9 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. Debug-.Invoke is the suggested method for starting a debugger.

The format of the command is:

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

The parameters for this command are:

- `Main_Unit => "<Image>"`: Specifies the name of the main program unit (the unit associated with the Main pragma) that will be debugged.
- `Options => ""`: Specifies what options are to be used with the command. The three options are:
  —`"Machine => Network Machine Name"`: Specifies the machine name given to the remote computer.

  —`"Directory => Os-9 Directory Name"`: Specifies the name of the directory that contains the executable module.

  —`Reuse_Debugger => True`: Specifies that the current debugger will be used. If set to false, a new debugger will be started and used.
- `Spawn_Job => True`: Specifies a Boolean value that determines whether the current job is spawned. The default is true.

For example, the following command debugs the Main_68k_Os9 program. The values for the remote machine and the remote directory are obtained from the library-switch file. The default is used for the Spawn_Job parameter. A new debugger is started and used:

```
Debug.Invoke(Main_Unit => "Main_68k_Os9",
             Reuse_Debugger => False);
```

9.1.1.2. M68k_Debugger

This command starts debugging of the selected main unit. Each time this command is used, a new debugger is started (the old debugger may still be active but not current). Optionally, the name of the remote machine and the directory on the remote machine that contains the transferred executable module in the OS-9 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:

```
M68k_Debugger(Main_Unit => "<Selection>",
              Options => "",
              Spawn_Job => True);
```

The parameters for this command are:

- `Main_Unit => "<Selection>"`: Specifies the name of the main program unit (the unit associated with the Main pragma) that will be debugged.
- `Options => ""`: Specifies what options are to be used with the command. The two options are:
  - —`"Machine => Network Machine Name"`: Specifies the machine name given to the remote computer.
  - —`"Directory => Os-9 Directory Name"`: Specifies the name of the directory that contains the executable module.
- `Spawn_Job => True`: Specifies a Boolean value that determines whether the current job is spawned. The default is true.

For example, the following command debugs the Main_68k_Os9 program. The values for the remote machine and the remote directory are obtained from the library-switch file. The default is used for the Spawn_Job parameter:

```
M68k_Debugger(Main_Unit => "Main_68k_Os9");
```

9.1.2. Determining Locations

The following commands are used to determine locations:

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

9.1.2.1. Debug.Address_To_Location

This procedure displays the Ada source-code location of the specified run-time 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, #3A4B (up to 8 characters—32 bits).

For example, the following command selects the source-code location corresponding to the memory address #FFFF3A4B:

```
Debug.Address_To_Location(Address => "#FFFF3A4B#);
```

9.1.2.2. Debug.Location_To_Address

This command displays the run-time 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 => "<Selection>"`: Specifies the selected source-code location.
- `Stack_Frame => 0`: Specifies the stack frame. The default is 0.

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

```
Debug.Location_To_Address
```

9.1.2.3. 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 => "<Selection>"`: Specifies the object (variable) whose location is to be determined.

- `Options => ""`: Specifies the options to be used with this command.

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

```
Debug.Object_Location;
```

9.1.3. Displaying Machine-Level Program Values

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

- Debug.Memory_Display

- Debug.Register_Display

9.1.3.1. 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 can be a hexadecimal number—for example, #3A4B 12AF# (up to 8 characters—32 bits)—or a name of a source location.

- `Count => 0`: Specifies the number of items to display.

- `Format => "Data"`: Specifies the format of the data to be displayed. The Format parameter must specify either "Code" or "Data": "Code" disassembles instruction memory; "Data" displays operand 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 (you should specify "Code").

For example, the following command displays 10 words of code starting at memory address #3A4B12AF:

```
Debug.Memory_Display(Address => "3A4B12AF",
                     Count => 10,
                     Format => Code);
```

9.1.3.2. 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 registers are displayed.

- `For_Task => ""`: Specifies the task to be used. If the null string (""), the default, is specified, the current task is used.

- `Stack_Frame => 0`: 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 => ""`: Specifies the format of the register data to be displayed. Format is not used.

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

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

9.1.4. Modifying Machine-Level Program Values

The following commands are used to modify memory or registers:

- Debug.Memory_Modify
- Debug.Register_Modify

9.1.4.1. Debug.Memory_Modify

This command is used to modify a word of memory.

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 => ">>Hex Address<<"`: Specifies the memory address to be modified. The address is a one- to eight-digit hexadecimal value (for example, #4A3B12AF).
- `Value => ">>Hex Value<<"`: Specifies the new value that is to be placed in the specified memory location. Value is also an eight-digit hexadecimal value (for example, #FFFFFFFF).
- `Width => 0`: Specifies the number of bits to be modified.
- `Format => "Data"`: Specifies the format of the data to be modified. The Format parameter can specify either "Code" or "Data": "Code" modifies operand memory; "Data" modifies instruction memory. You can only modify data within your address space. Other references will indicate an error.

For example, the following command places the value FFFFFFFF in the memory location 4A3B12AF, which is instruction memory:

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

9.1.4.2. 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 => ">>REGISTER NAME<<"`: Specifies the number of the register (for example, D2).

- `Value => ">>HEX VALUE<<"`: Specifies an 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 => 0`: 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 => ""`: Specifies the format of the data to be modified. The Format parameter can specify either "Code" or "Data": "Code" modifies operand memory; "Data" modifies instruction memory. You can only modify data within your address space. Other references will indicate an error.

For example, the following command places the value FFFFFFFF in the register D2; the format is instruction memory:

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

9.1.5. Additional Debug Commands

In addition to the above commands, the following commands are used with the M68K cross-debugger:

- Debug.Current_Debugger

- Debug.Kill

- Debug.Run

9.1.5.1. Debug.Current_Debugger

This command is used to establish an M68K cross-debugger as the current debugger (if the M68K cross-debugger already exists).

Making a debugger the current debugger means that debugging commands will now control the execution of the current debugging job.

Running a debugger Command window directly off an M68K cross-debugger window, or pressing a debugger key while in that window, makes the M68K cross-debugger the current debugger.

Running Debug.Invoke on a Motorola_68k unit also makes the M68K 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 M68K cross-debugger.

For example, the following command sets the current debugger to an M68K cross-debugger:

```
    Debug.Current_Debugger(Target => "Motorola_68k");
```

9.1.5.2. Debug.Kill

This command terminates an M68K cross-debugger session.

If the Debugger parameter is not set to true, the OS-9 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 => True`: Specifies a Boolean value that determines whether the the OS-9 program will be killed.

- `Debugger => True`: Specifies whether the debugger will be killed.

For example, the following command kills both the OS-9 job and the M68K cross-debugger:

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

9.1.5.3. Debug.Run

This command can be used to provide machine-level stepping. 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 => Debug.Statement`: Specifies the machine instruction at which machine-level stepping will commence.
- `Count => 1`: Specifies the number of machine-level steps to take.
- `In_Task => ""`: Specifies the task in which the machine-level stepping is to occur.

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

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

9.2. Commands Used with Debuggers

Table 9-2 lists the commands found in package !Commands.Common that are used when debugging programs. These commands are used with both the R1000 native debugger and the M68K 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

Debugging 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.

Table 9-2 Package Common Debugging Commands (continued)

Debugging Command	Function
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.

Table 9-3 lists the commands found in package !Commands.Debug that are used when debugging programs. These commands are used with both the R1000 native debugger and the M68K cross-debugger. For more information on these commands, consult the Debugging (DEB) book of the *Rational Environment Reference Manual.*

Table 9-3 Package Debug Debugging Commands

Debugging 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 9-3 Package Debug Debugging Commands (continued)

Debugging 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-dependent 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.
Kill	Kills the job being debugged and/or the debugger for the session.
Location_To_Address	Displays the code-segment address for the machine instructions associated with the specified location.
Memory_Display	Displays the contents of absolute memory.
Modify	Modifies or changes the values of the specified object.
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.

Table 9-3 Package Debug Debugging Commands (continued)

Debugging Command	Function
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.
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 9-4 lists the commands found in package !Commands.Debug_Tools that are used when debugging programs. These commands are used with both the R1000 native debugger and the M68K cross-debugger. For more information on these commands, consult the Debugging (DEB) book of the *Rational Environment Reference Manual.*

Table 9-4 Package Debug_Tools Debugging Commands

Debugging Command	Function
Ada_Location	Returns a string representing the Ada name of the source location in the calling subprogram or a caller of the calling subprogram.
Debug_Off	Disables debugging in the calling task.
Debug_On	Enables debugging for the calling task.
Debugging	Returns true if the currently executing program is under the control of the debugger; otherwise, the command returns false.
Get_Exception_Name	Returns the name of the most recently raised exception for the task that calls this function; optionally, the command returns additional machine-related information about the exception.
Get_Release_Location	Returns a representation of the location in the source from which the most recent exception for the task calling this function was raised; optionally, the command returns additional machine- related information about the raised location.
Get_Task_Name	Returns any task name set by the Set_Task_Name procedure.
Image	Returns the string that is the image of the value of the Value parameter for the special display for T.
Message	Causes a message to be displayed in the Debugger window.
Register	Provides facilities that enable the user to write *special display* routines for the debugger; the debugger uses the routines to display the value of variables and to perform other actions.
Set_Task_Name	Assigns a string *nickname* for the named task.
Un_Register	Causes a special display registered for a type to be used no longer by the debugger for displaying objects of that type.
User_Break	Causes the calling task to stop as though a breakpoint were reached.

9.3. Differences between the R1000 and the M68K Cross-Debugger

The differences between the R1000 debugger and M68K cross-debugger are discussed in this section.

9.3.1. Breakpoints

On the M68K, 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.

9.3.2. Exceptions

The M68K cross-compiler and run-time system does not support flavors of exceptions.

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 M68K 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 run times 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.

9.3.3. Elaboration

To elaborate a program on the M68K, 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 and breakpointing and other operations that depend on task name behave differently during elaboration.

9.3.4. Object Evaluation

On the M68K, no elaboration check is performed by the debugger when it displays an object. Data displayed 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.

9.3.5. Memory Display

The M68K 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 M68K debugger.

9.3.6. Stack Frames

Block statements and accept statements are inlined by the M68K cross-compiler. They are displayed as separate frames (as on the R1000 debugger) even though no physical frame exists.

10. Configuration Management and Version Control

The M68K/OS-9 Cross-Development Facility provides the user with the Rational Environment's resources for configuration management and version control (CMVC). These resources can be used to accomplish the following:

- Project partitioning: The user can break a project into a manageable number of higher-level components called *subsystems*, each containing a group of logically related objects. For Ada units, subsystems are units of decomposition similar to, but larger than, the Ada package, which preserves on a larger scale the Ada notion of separate specification and implementation.

- Version control: The user can control and track changes to individual objects within each subsystem, determine which versions can be changed and who can change them, and record why the changes were made.

- Configuration management: The user can construct, release, and maintain multiple consistent sets (or *configurations*) of versions within each subsystem. (Each alternative configuration constitutes a *view* of the subsystem.) At a higher level, configuration management refers to combining views from each subsystem in order to build entire programs.

10.1. CMVC Review

This section presents an overview of configuration management and version control. For details on commands, consult the Project Management (PM) book of the *Rational Environment Reference Manual.* For an example of CMVC and subsystem use, see "Using Motorola_68k Subsystem Views" in Chapter 3.

10.1.1. Issues of Project Management

Although using worlds or directories can make it easier to understand the high-level structure of large projects, their use cannot solve the following problems in project management:

- Project size: When there are too many program units, it can be difficult to:

  —Reason about the program's overall design

  —Keep track of the dependencies among units

  —Allocate well-defined portions of the program to individual developers or development teams

- Program dependencies: When there are too many dependencies, making changes can be time-consuming because the changes must be verified by recompiling the changed units and all of their direct and indirect dependents. These recompilation dependencies also make it difficult for developers and teams to work and test in parallel, because a change in one team's portion of the program may entail recompilation of another team's portion.
- Design degradation: Regardless of the size of the program, it is difficult to prevent design degradation resulting from the introduction of unwanted dependencies between units.
- Version control: Preserving previous versions of units and controlling access to shared units can be cumbersome.

A more powerful kind of Environment library structure, the *subsystem*, can be used instead to express and enforce program design.

10.1.2. Subsystems

Subsystems encapsulate a program's compilation units into higher-level components, just as Ada packages encapsulate related subprograms, type declarations, and the like. Depending on its size, each subsystem can be assigned to individual developers or to a team of developers.

Subsystems are more powerful than other libraries for the following reasons:

- They provide a mechanism for defining and limiting interfaces among the program components they encapsulate.
- They provide a mechanism for developing alternative implementations of program components. Execution and testing is a matter of specifying the desired combination of precompiled implementations, one from each subsystem.
- CMVC operations are available only within subsystems.

10.1.3. Version Control

When a program component is encapsulated within a subsystem, individual objects in the component can be *controlled*—that is, made subject to version control. Controlled objects must be *checked out* to be modified; checking out an object reserves it for editing by acquiring the object's *reservation token*. When desired, the modified object can be *checked in*.

Every subsystem contains a *CMVC database* that records changes made to each controlled object. Each time an object is checked out and then checked in, a new *generation* of the object is created in the CMVC database. Therefore, the CMVC database records the successive generations of each controlled object within a program component.

10.1.4. Configurations

Objects in subsystems reside in program libraries and therefore can be compiled using the normal Environment mechanisms. Each subsystem contains at least one *working* library from which units can be checked out, modified, checked in, compiled, and tested.

When a particular combination (or *configuration*) of object generations compiles satisfactorily, a *release* of that configuration is made. Each release is a frozen copy of the working library and therefore is a full program library itself. Successive releases can be thought of as "snapshots" or "views" of the subsystem through time. Accordingly, the release libraries and working libraries within a subsystem are called *views* (more specifically, *release views* and *working views*). A series of releases created from a single working view is called a development *path*.

It is important to keep in mind that each view is:

- A source configuration, in that it specifies a particular generation for each object in the subsystem

- A program library, in that it enforces Ada semantic consistency among the specified generations

Therefore, CMVC operations integrate configuration management with library management.

10.1.5. Interfaces among Subsystems

Interfaces can be defined between subsystems using different kinds of views.

- Load view: The working and released views are *load views*. Each load view contains a full implementation of the program component that is encapsulated in the subsystem.

- Spec view: A second kind of view, called the *spec view*, can be created to define the set of implemented units that are potentially available, or visible, to units in other subsystems. Spec views thus define a subsystem's *exports*; as such, spec views can be *imported* by *client* views in other subsystems. When a client view from one subsystem imports a spec view from another subsystem, dependencies can be set up among units from the two subsystems. Subsystem imports and exports thus enforce design decisions, because *with* clauses can reference only units from imported views.

- Combined view: A third kind of view, called the *combined view*, is similar in contents to the load view. However, it contains both exported specifications and their implementation and also may be imported. Combined views do not require an activity for execution. Combined views do have working and release forms. See the section on "Differences between R1000 and M68K/OS-9 Subsystems," in this chapter, for a discussion of the limitations of combined views.

Subsystem interfaces are analogous to Ada package interfaces, as follows:

- A spec view is analogous to an Ada package specification, which defines the resources that are available to client units.

- A load view and a combined view are analogous to a package body, which implements the resources promised by the specification.

- The import relation is analogous to a *with* clause, which enables a client unit to actually use the specified resources.

Because only spec views can be imported, client views compile against spec views, not load views. Therefore, units in a working load view can be changed without requiring recompilation of any other views, provided that the working views remain *compatible* with the spec view that defines its exports.

Compatibility allows a load view to differ in certain specific ways from the spec view that represents it. One important kind of change that preserves compatibility is to change the private part of an exported unit. If such a change is made to a load view, no change or recompilation is required of the spec view or any of its client views. In this way, subsystem interfaces make *closed private parts* possible.

By buffering recompilation for many kinds of changes, subsystem interfaces enable subsystems to be developed in parallel—that is, a team of developers can change and test the implementation in its own subsystem without necessarily causing recompilation elsewhere. However, changes that do not preserve compatibility require modification of both spec and load views; a changed spec view can affect client views in other subsystems.

10.1.6. Program Execution

A subsystem typically consists of at least one spec view, against which client views are compiled, and at least one load view, which contains the units that are actually executed. As periodic releases are made from the working load view, a single subsystem accumulates multiple load views, each implementing the resources specified in a given spec view.

To execute the program composed of such subsystems, an execution table, called an *activity*, must be set up to specify which of the alternative load views is to be used with the spec view that is imported from each subsystem. Because only one spec view can be imported from a given subsystem, the activity contains exactly one entry for each subsystem that is required for execution. Activities thus can be thought of as specifying configurations of views, which contain configurations of generations. Note that subsystems with combined views do not require activities. (See the section on "Differences between R1000 and M68K/OS-9 Subsystems," in this chapter, for a discussion of the use of combined views.)

Activities provide a flexible means of constructing programs from alternative subsystem implementations. Any number of activities can be created; recombinant testing is a matter of editing an activity to specify precompiled views rather than recompiling an entire program from scratch. (Note that each entry must specify a compatible pair of load and spec views.)

By choosing appropriate views from each subsystem, the user can construct system tests that isolate the effects of specific changed views. For example, to test a new implementation of a particular subsystem, an activity typically specifies the working view from that subsystem, along with stable baseline releases from the other subsystems in the program. The availability of both released and working views enables subsystem testing to proceed in parallel, because each team can continue ongoing work in the working view while other teams are testing against frozen releases.

10.1.7. Parallel Development within Subsystems

Parallel development is also possible within subsystems as well as between them. When a team is assigned to implement a subsystem, a separate *subpath* can be created for each individual on the team. Subpaths are working views in which changes can be made and tested. They are created as full copies of the main path's working view.

Editing can proceed without conflict because controlled objects are *joined*. Each joined object shares a reservation token with the corresponding objects in the other subpaths; a given joined object can be checked out in only one subpath at a time. In this way, a single set of generations is maintained for multiple copies of an object.

A subpath can become out of date when objects are checked out and modified in other subpaths. Objects in a subpath can be brought up to date either by checking them out or by *accepting changes* from the latest generation into that view.

Developers working in two subpaths can access an object concurrently if it is *severed*. Severing provides separate copies of an object with their own reservation tokens, so that both copies can be checked out independently. Separate sets of generations are kept for severed objects. Severed objects can be synchronized by *merging changes* from one object into the other.

Separate development efforts within a single subsystem can be maintained in alternative development paths. For example, maintenance of an existing program can continue in one path while development of the next major field release proceeds in another path.

10.1.8. Project Reporting

Information about a project can be gathered in several ways. Each generation of every controlled object has *notes* associated with it, which can be used as a "scratchpad" for arbitrary comments. Comments from checkout and checkin commands are automatically entered in an object's notes. A scratchpad for notes also is associated with each release, and it automatically logs any comments that are specified when a release is made.

For more comprehensive project reporting, *work orders* can be used to define and assign units of work. When development proceeds in response to a given work order, time-stamped comments are logged into the work order whenever commands from package Cmvc are executed. In addition, information can be entered in user-defined fields on each work order.

Work orders can be grouped for easy reference using *work-order lists*. For example, a work-order list can contain entries for all work orders assigned to a given user or for all work orders that have been closed.

All the work orders for a given project or subproject are created from a single template called a *venture*. User-defined fields are created in ventures; ventures also determine *policies* that govern the work done in response to work orders. For example, a policy can prevent a CMVC command from executing unless the parameter for comments is filled in.

User-generated tools can be used to create reports from the information gathered by work orders.

10.1.9. Multihost, Multisite Development

When a program is partitioned into subsystems, it can be developed on multiple R1000s, either at the same site or at different geographic sites. Such development accommodates very large programs, especially when program components have been assigned to subcontractors.

When multiple R1000s are used, each one hosts a copy of every subsystem in the program. However, only one copy of a given subsystem, called the *primary subsystem*, can support ongoing development. The other copies, called *secondary subsystems*, are essentially local copies for execution and test.

Typically, each R1000 hosts a primary subsystem and some number of secondary subsystems. When a new release is made in the primary subsystem, that release can be copied, via network or tape, into the corresponding secondary subsystems on each of the other R1000s. On each R1000, the copied release then can be compiled with the releases from the other subsystems and the program can be executed. Note that, instead of releasing the source for a load view, a *code view* can be made and copied from the primary to secondary subsystems. A code view contains only the executable code from the compiled load view.

10.2. Differences between R1000 and M68K/OS-9 Subsystems

When possible, M68K/OS-9 subsystems should be constructed using spec and load views. However, although the use of subsystems is similar for R1000 and M68K/OS-9 subsystems, there are some features of M68K/OS-9 subsystems of which the user should be aware.

10.2.1. Combined Views

Rational uses code-shared generics, whereas M68K/OS-9 systems use macro expansion with their generics. Therefore, if a program exports generics, combined views must be used in the subsystem. However, there are some limitations to the use of combined views:

- Since combined-view exports are directly imported, compilation obsolescence propagates to importing views. Therefore, a change in one view can require recompilation in all other views that import the changed view. This recompilation can be extensive and time-consuming in large programs with many subsystems and views.

- *Closed private parts* are not available.

- Because execution with a new release of a combined view requires importing the new view, recompilation may be required. When the new view is imported, the old view must be removed completely from the import-closure *consistent semantic network*. Therefore, a released combined view cannot import working combined views.

10.2.1.1. Releases of Combined Views

Integration of combined views, which is similar to integration of spec views, requires that:

- A single semantic network be maintained

- The imports be changed simultaneously for all clients

Releasing combined views differs from releasing spec/load views. Released views are frozen and cannot be modified. Therefore, non-upward-compatible changes cannot be made to imported views. Testing the subsystem thus is less flexible than with spec/load views.

When using the Cmvc.Release command:

- The From_Working_View parameter accepts a list of views to release at the same time.

- The imports now automatically point to the other released views in the new set.

10.2.1.2. Change Propagation in Combined Views

Changes made in the code of combined views propagate in source form. Incremental changes made in one combined view are not propagated incrementally to other views. If a change is made that requires demotion to the source state, the units in the dependency closure also are demoted to source.

Incremental changes are possible, but they must be done manually—that is, a change made in one view also must be made in the importing view; otherwise, the views are demoted to source.

10.2.2. Code Views

Another important difference in M68K/OS-9 subsystems is that *code* views cannot be created.

10.3. CMVC Commands

The more frequently used CMVC commands are listed here. These commands are found in the following four packages:

- Activity
- Cmvc
- Cmvc_Maintenance
- Work_Order

Package Work_Order contains the following three packages as well as its functions and procedures:

- Editor
- List_Editor
- Venture_Editor

The commands used with CMVC and their functions are listed here in tabular form. For a complete description of the CMVC commands, consult the Project Management (PM) book of the *Rational Environment Reference Manual.*

10.3.1. Package !Commands.Activity

Package Activity provides operations for creating, viewing, and manipulating activities and for identifying which activity is the current activity for a running job or session.

In addition to the commands relating to activities, an editor provides editing operations specific to activities. Many of the operations in package !Commands.Common apply to activities. Table 10-1 lists the commands found in package Common that apply to activities.

Table 10-1 Package Common Commands for Editing Activities

Command	Purpose
Abandon	Ends editing of the activity and removes the window from the screen. Because all changes to activities are not made permanent until committed, any uncommitted changes will be lost.
Commit	Makes permanent any changes made to the activity.
Create_Command	Creates a Command window below the current window. The *use* clause in the Command window includes package Activity, so operations in package Activity are directly visible without qualification in the Command window.
Definition	Finds the definition of the subsystem corresponding to the selected entry on which the cursor resides. This procedure creates a window containing that subsystem.
Edit	Prompts the user for changes to the selected entry, or to the entry on which the cursor resides when the command is given, by creating a Command window and placing in it the command: `Change(Spec_View => "", Load_View => "");` The user fills in one or both parameters, as desired.
Release	Makes any changes to the activity permanent, releases control of (unlocks) the activity, and then destroys the window.
Sort_Image	Sorts the activity image according to the specified format.
Object.Child	Selects the entry in the activity on which the cursor resides.
Object.Delete	Deletes the selected entry on which the cursor resides.
Object.Elide	Controls the level of detail displayed in the image of the current activity.
Object.Expand	Controls the level of detail displayed in the image of the current activity.
Object.Explain	Uncompresses a subsystem entry, listing each component (subsystem name, spec view, and load view) of the entry on separate lines.
Object.First_Child	Selects the first entry of the activity.

Table 10-1 Package Common Commands for Editing Activities (continued)

Command	Purpose
Object.Insert	Inserts a new subsystem entry or modifies an existing entry in the activity by prompting the user. This command creates a Command window and places in it the following command: `Insert (Subsystem => "", Spec_View => "", Load_View => "");` The user fills in the details as desired.
Object.Last_Child	Selects the last entry of the activity.
Object.Next	Selects the next entry in the activity if an entry is selected. If no entry is selected, this commands selects the entry on which the cursor currently resides.
Object.Parent	Selects the entry in the activity on which the currently resides. If an entry is already selected, the procedure selects all entries in the activity.
Object.Previous	Selects the previous entry in the activity if an entry is selected. If no entry is selected, the procedure selects the entry on which the cursor currently resides.

Table 10-2 lists the commands found in package !Commands.Activity.

Table 10-2 Package Activity Commands

Command	Purpose
Add	Modifies the activity specified by The_Activity parameter by updating an existing entry for a subsystem or by adding a new entry if an entry for the specified subsystem does not already exist.
Change	Modifies the spec-view and/or load-view components of the currently selected subsystem entry on which the cursor currently resides.
Create	Creates a new activity.
Current	Displays the name of the activity that is associated with the current job.
Display	Displays an image of the specified activity.
Edit	Invokes the activity object editor on the specified target activity.

Table 10-2 Package Activity Commands (continued)

Command	Purpose
Enclosing_Subsystem	Displays the name of the corresponding subsystem for the specified view.
Enclosing_View	Displays the name of the view that contains the specified unit.
Insert	Modifies an activity to update an existing entry for a subsystem or adds a new entry if one does not already exist for the specified subsystem.
Merge	Copies into the specified target those subsystem entries defined in the source activity that match the pattern specified in the Subsystem, Spec_View, and Load_View parameters.
Nil	Returns the name of an empty activity.
Remove	Deletes a subsystem entry from an activity.
Set	Changes the current activity for the running job to the specified activity.
Set_Default	Makes the specified activity the current activity for the current session.
Set_Load_View	Modifies the load view for the specified subsystem entry in The_Activity parameter.
Set_Spec_View	Modifies the spec view for the specified subsystem entry in The_Activity parameter.
The_Current_Activity	Returns the name of the current activity associated with the running job.
The_Enclosing_Subsystem	Returns the name of the subsystem that contains the current view.
The_Enclosing_View	Returns the name of the view that contains the specified unit.
Visit	Invokes the activity editor on the specified activity and replaces the old activity if one is currently being edited.
Write	Copies the contents of an activity window into a new activity in the directory system.

10.3.2. Package !Commands.Cmvc

Package Cmvc provides operations for creating, viewing, and manipulating subsystems as well as creating and controlling paths within the subsystem.

Table 10-3 lists the commands found in package !Commands.Cmvc.

Table 10-3 Package Cmvc Commands

Command	Purpose
Abandon_Reservation	Abandons the reservation on one or more checked-out objects, effectively canceling the checkout of those objects.
Accept_Changes	Updates the object(s) specified in the destination parameter to the generation(s) indicated by the Source parameter; that is, the destination objects are changed to reflect any modification that have been made to the corresponding source objects.
Append_Notes	Appends the specified string to the end of the notes for the specifed controlled object. The notes for a controlled object are stored in the CMVC database.
Build	Builds views from the specified configuration objects. Views corresponding to the specified configurations must not exist.
Check_In	Releases the reserved right to update the specified object or set of objects and stores the text of the new generation(s) in the CMVC database.
Check_Out	Reserves the right to modify the specified object(s).
Copy	Creates one or more new views by copying specified view(s). By default, the Copy command makes new working load or combined views, depending on the kind of views named in the From_View parameter. Copy can also be used to make spec views by setting the Create_Spec_View parameter to true.
Create_Empty_Note_Window	Creates an empty window for the purpose of composing notes for the specified controlled object. The notes for a controlled object are stored in the CMVC database.
Destroy_Subsystem	Destroys the specified subsystem(s).
Destroy_View	Destroys the named view(s) and all of their subdirectory structures, including the Ada units in the Units directories.
Get_Notes	Retrieves the notes for the specified controlled objects. The notes are retrieved from the CMVC database and are either displayed in a window or written into a file.
Import	Imports the specified spec or combined views into the designated view(s).
Import_Views	Returns a naming string that names all the views that are imported by the specified view.
Information	Displays various kinds of information about the specified view in the output window.
Initial	Builds a new subsystem of the specified type—namely, spec/load or combined.

Table 10-3 Package Cmvc Commands (continued)

Command	Purpose
Join	Joins the specified controlled objects to the corresponding objects in the designated view. When views are joined across views, they form a join set. Objects in a join set have the same simple name and share a reservation token, so that only one object in the set can be checked out at a time.
Make_Code_View	Makes a code view from the specified load view.
Make_Controlled	Makes the specified object(s) controlled by the CMVC system and therefore subject to reservation.
Make_Path	Creates a copy of each of the specified views, starting new development paths.
Make_Spec_View	Creates a new spec view from each of the specified views in a spec/load subsystem.
Make_Subpath	Creates a copy of each of the specified views in order to start new development subpaths.
Make_Uncontrolled	Makes the specified objects uncontrolled, so that change information about them is no longer recorded in the CMVC database.
Merge_Changes	Merges two instances of the same object that are located in two different working views.
Put_Notes	Replaces the notes for the specified controlled object with the contents of the specified file. The notes for a controlled object are stored in the CMVC database.
Release	Creates a new released view from each of the specified working views.
Remove_Import	Removes the links that were created when the view specified by the View parameter was imported.
Remove_Unused_Imports	Removes imports from the specified view(s) if none of the links created by those imports are needed for compilation. Links are removed only on an import-by-import basis.
Replace_Model	Replaces the model world for the specified view.
Revert	Reverts the specified object(s) to the specified generation.

Table 10-3 Package Cmvc Commands (continued)

Command	Purpose
Sever	Severs the specified objects from their respective join sets. When an object is severed, it is given a different reservation token.
Show	Displays checkout and generation information for the specified controlled objects.
Show_All_Checked_Out	Displays a list of the objects specified in the view that are checked out.
Show_All_Controlled	Lists the controlled objects in the specified view(s).
Show_All_Uncontrolled	Lists any of the specified objects that are uncontrolled.
Show_Checked_Out_By_User	Lists the objects in the specified view(s) that are checked out by the specified user.
Show_Checked_Out_In_View	Lists the objects that are checked out in the specified view(s), regardless of who checked them out.
Show_History	Displays the history for the specified view or object within a view.
Show_History_By_Generation	Displays the history for one or more controlled objects across the specified range of generations.
Show_Image_Of_Generation	Reconstructs an image of the specified generation of the designated object.
Show_Out_Of_Date_Objects	Lists the objects in the specified view(s) that are not at the most recent generation.

10.3.3. Package !Commands.Cmvc_Maintenance

Package Cmvc_Maintenance provides operations for maintaining the subsystem.

Table 10-4 lists the commands found in package !Commands.Cmvc_Maintenance.

Table 10-4 Package Cmvc_Maintenance Commands

Command	Purpose
Check_Consistency	Checks the consistency of the specified views with respect to the CMVC database and the Environment library system.
Convert_Old_Subsystem	Converts the views in one or more subsystems from the Gamma format to the Delta format so that CMVC operations can be used.
Destroy_Cdb	Destroys the compatibility database for the specified subsystem.
Display_Cdb	Displays information from the CMVC compatibility database for each of the specified subsystems.
Display_Code_View	Displays information about the specified code view. By default, the command displays a list of the units in the code view.
Expunge_Database	Expunges the CMVC database, removing stored information and history about unused configurations or objects.
Make_Primary	Converts the specified secondary subsystem into a primary subsystem with its own updatable compatibility database.
Make_Secondary	Converts the specified primary subsystem into a secondary subsystem with a read-only compatibility database.
Repair_Cdb	Verifies that the information in the specified subsystem's compatibility database is consistent with the DIANA representation of the subsystem's compiled units.
Update_Cdb	Updates a secondary subsystem's compatibility database by copying the compatibility database from another subsystem. The two subsystems must have the same R1000 subsystem identification number, although they may be on different R1000s.

10.3.4. Package !Commands.Work_Order

Package Work_Order provides operations for creating and using work orders.

Table 10-5 lists the commands found in package !Commands.Work_Order.

Table 10-5 Package Work_Order Commands

Command	Purpose
Add_To_List	Adds one or more work orders to a work-order list.
Close	Sets the status of the specified work order to closed. Once a work order has been closed, it no longer can be modified.
Create	Creates a work order on the specified venture and adds it to the work-order list.
Create_Field	Defines a new user-defined field with the designated type in the specified venture.
Create_List	Creates a work-order list on the specified venture.
Create_Venture	Creates a new venture.
Default	Returns the name of the user's default work order in the specified venture.
Default_List	Returns the name of the user's default work-order list in the specified venture.
Default_Venture	Returns the pathname of the default venture for a user.
Display	Displays the contents of the specified work order in the output window.
Display_List	Displays the contents of the specified work-order list in the output window.
Display_Venture	Displays the contents of the specified venture in the output window.
Edit	Edits the designated work order. If a window already exists for that work order, the window is reused. From the window, the work order can be edited with the operations from package !Commands.Common that apply to this class of object.
Edit_List	Edits the designated work-order list. If a window already exists for that work-order list, the window is reused. From the window, the work-order list can be edited with the operations from package !Commands.Common that apply to this class of object.
Edit_Venture	Invokes the venture object editor for the designated venture. The procedure creates a window in which the designated venture is displayed. If a window already exists for that venture, the window is reused. From the window, the venture can be edited with the operations from package !Commands.Common that apply to this class of object.

Table 10-5 Package Work_Order Commands (continued)

Command	Purpose
Notes	Returns the notes field of the specified work order.
Notes_List	Returns the notes field of the specified work-order list.
Notes_Venture	Returns the notes field for the specified venture.
Remove_From_List	Removes the entry for the specified work order(s) from a work-order list.
Set_Default	Sets the specified work order to be the default for a given user and session whenever the work order's parent venture is the default. This command modifies the venture by adding or changing the mapping from session to work order in the specified venture.
Set_Default_List	Sets the specified work-order list to be the default for a given user and session whenever the work-order lists's parent venture is the default. This command modifies the venture by adding or changing the mapping from session to work order in the specified venture.
Set_Default_Venture	Sets the default venture for the specified session. Setting a venture to be the default automatically sets the default work order and the default work-order list for the current session, if such defaults have been specified for that venture. Setting a default venture with this command automatically sets the value of the Cmvc.Default_Venture session switch to the specified venture name.
Set_Notes	Modifies the notes field for the specified work order. Any existing notes in the specified work order are replaced by the new notes.
Set_Notes_List	Modifies the notes field for the specified work-order list. Any existing notes in the specified work-order list are replaced by the new notes.
Set_Venture	Modifies the notes field for the specified venture. Any existing notes in the specified venture are replaced by the new notes.
Set_Venture_Policy	Sets the specified venture policy switch to the specified value. This command also can be used to determine the value of a particular switch for a venture that is not currently displayed.
Editor.Add_Comment	Adds a comment to those recorded in the work order. Once added, a comment cannot be removed.
Editor.Add_Configuration	Adds a configuration to those recorded in the work order. Once added, a configuration cannot be removed.

Table 10-5 Package Work_Order Commands (continued)

Command	Purpose
Editor.Add_User	Adds a user session to those recorded in the work order. Once added, a user cannot be removed.
Editor.Add_Version	Adds a version to those recorded in the work order. Once added, a version cannot be removed.
Editor.Set_Field	Sets the Boolean value of the specified work-order field to the specified value. Once set, a work-order field cannot be modified.
Editor.Set_Field	Sets the integer value of the specified work-order field to the specified value. Once set, a work-order field cannot be modified.
Editor.Set_Field	Sets the string value of the specified work-order field to the specified value. Once set, a work-order field cannot be modified.
Editor.Set_Notes	Sets the notes field of the work order to the specified string. The specified text replaces the existing text.
List_Editor.Add	Adds the specified work orders to the work-order list.
List_Editor.Set_Notes	Sets the notes field of the work-order list to the specified string. The specified text replaces the existing text.
Venture_Editor-.Set_Default_List	Sets the default work-order list for a specified user session on the local venture. Each session can have a different work-order list.
Venture_Editor-.Set_Default_Order	Sets the default work order for a specific user session on the local venture. Each user session can have a different default work order.
Venture_Editor.Set_Field_Info	Sets the numeric tag of a user-defined field and specifies whether that field is modifiable. Numeric tags control the relative display position of the field within the venture.
Venture_Editor.Set_Notes	Sets the notes field of the venture to the specified string. The specified string replaces the existing text.
Venture_Editor.Set_Policy	Sets the specified policy switch to the specified value.
Venture_Editor.Spread_Fields	Renumbers all user-defined fields, assigning new numeric tags using the specified interval.

11. Appendix F for the Rational M68K/OS-9 Cross-Development Facility

The *Reference Manual for the Ada Programming Language* 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 chapter of the *Rational M68K/OS-9 Cross-Development Facility* manual is the Appendix F for the M68K/OS-9 cross-compiler. It contains sections that describe the following implementation-dependent features:

- Implementation-dependent pragmas
- Implementation-dependent attributes
- Package Standard
- Package System
- Restrictions on representation clauses
- Names denoting implementation-dependent components
- Interpretation of expressions that appear in address clauses
- Unchecked conversion
- Machine code

Implementation-Dependent Pragmas

The M68K/OS-9 cross-compiler supports pragmas for application software development in addition to those listed in Appendix B of the *Reference Manual for the Ada Programming Language*. They are described below, along with additional clarifications and restrictions for pragmas defined in Appendix B of the *Reference Manual for the Ada Programming Language*.

- Main: A parameterless library unit procedure can be designated as a main program by including a Main pragma at the end of the specification or body of the unit. This pragma causes the linker to run and creates an executable program when the body of this subprogram is coded.

  The pragma has the following optional parameters:

  —Stack_Size => <static integer expression>: Specifies the size of the main task stack in storage units; if not specified, a default of 4K words is used.

  —Heap_Size => <static integer expression>: Specifies the size of the heap in storage units; if not specified, a default of 16K words is used.

—Priority => <static integer expression>: Specifies the priority of the main program; if not specified, a default of 127 is used.

- Suppress_All: This pragma must appear immediately within a declarative part. It is equivalent to the following sequence of pragmas:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Range_Check);
pragma Suppress (Division_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Storage_Check);
```

Note that the Suppress_All pragma cannot prevent the raising of certain exceptions. For example, numeric overflow is detected by hardware, which results in the predefined Numeric_Error exception being raised. Refer to Chapter 7, "Run-Time Organization," for more information.

- Import_Object: This pragma allows Ada units to reference objects declared in non-Ada units. The user of this pragma is responsible for understanding the representation of the object being imported. Only objects at static scopes—that is, not within any task or subprogram—can be designated by this pragma. Parameters are:

—The Ada name that is used to reference the object being imported.

—The link name of the object that is being imported.

The pragma must appear in the same declarative part as the named Ada unit, which must be directly visible at the point of the pragma.

References to the named Ada unit are translated by the cross-compiler into references to the link name. The link name must be exported by another module of the program. The following example shows a variable (Imported_Variable) of type Integer, which is being imported from another module; the link name associated with the variable is XYZ:

```
Imported_Variable : Integer;
pragma Import_Object(Imported_Variable, "XYZ");
```

Note that the cross-compiler does not ensure that the syntax of the link-name string is consistent with that required by the assembler and the linker.

- Export_Object: This pragma allows Ada units to be exported to non-Ada objects. The user of this pragma is responsible for understanding the representation of the unit being exported. Only units at static scopes—that is, not within any task or subprogram—can be designated by this pragma. Parameters are:

  —The Ada name that is used to reference the object being exported.

  —The link name of the object that is being exported.

  The pragma must appear in the same declarative part as the named Ada unit, which must be directly visible at the point of the pragma.

  A global symbol is created to represent the address of the storage allocated to the Ada unit. Non-Ada modules reference the link name of the object. The following example shows a variable (Exported_Variable) of type Integer, which is being exported to another module; the link name associated with the variable is ABC:

  ```
  Exported_Variable : Integer;
  pragma Export_Object(Exported_Variable, "ABC");
  ```

  Note that the cross-compiler does not ensure that the syntax of the link-name string is consistent with that required by the assembler and the linker.

- Import_Procedure: This pragma allows an Ada unit to call a non-Ada procedure. This pragma supplements the Interface pragma and is used to specify the interface to such an imported subprogram. The user of this pragma is responsible for ensuring that all requirements for the called subprogram are met. The following example shows a procedure (Imported_Procedure) that is an external subprogram being imported:

  ```
  procedure Imported_Procedure(X : Integer;  Y : out Float);
  pragma Import_Procedure (Imported_Procedure, "PQR");
  ```

- Import_Function: This pragma allows an Ada unit to call a non-Ada function. This pragma supplements the Interface pragma and is used to specify the interface to such an imported subprogram. The user of this pragma is responsible for ensuring that all the requirements of the called subprogram are met. The following example shows a function (Imported_Function) that is an external subprogram being imported:

  ```
  procedure Imported_Function(X : Integer) return Float;
  pragma Import_Function (Imported_Function, "STU");
  ```

The complete syntax for these pragmas is:

```
pragma IMPORT_FUNCTION
          (internal_name
           [, external_name]
           [, function_profile]
            , mechanisms)
```

```
pragma IMPORT_PROCEDURE |
       IMPORT_VALUED_PROCEDURE
            (internal_name
             [, external_name]
             [, procedure_profile]
              , mechanisms)

pragma EXPORT_FUNCTION
            (internal_name
             [, external_name]
             [, function_profile])

pragma EXPORT_PROCEDURE
            (internal_name
             [, external_name]
             [, procedure_profile])


pragma IMPORT_OBJECT |
       EXPORT_OBJECT
            (internal_name
             [, external_name]
             [, size])

internal_name     ::= [INTERNAL_NAME =>] designator

external_name     ::= [EXTERNAL_NAME =>] string_literal

function_profile  ::= param_profile, result_profile | nickname

procedure_profile ::= param_profile | nickname

param_profile     ::= [PARAMETER_TYPES =>] parameter_types

result_profile    ::= [RESULT_TYPE =>] type_mark

parameter_types   ::= (null) | (type_mark {, type_mark})

nickname          ::= [NICKNAME =>] identifier | integer_literal

mechanisms        ::= [MECHANISM =>] mechanism | (mechanism
                                                 {, mechanism})

mechanism         ::= VALUE | REFERENCE
```

Implementation-Dependent Attributes

There are no implementation-dependent attributes.

Package Standard

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 -3.40282346638529E+38 .. 3.40282346638529E+38;
    type Long_Float is digits 15
               range -1.79769313486231E+308 .. 1.79769313486231E+308;
    type Duration is delta 9.76562500000000E-04
               range -1.31072000000000E+05 .. 1.31071999023437E+05;
    subtype Natural is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;

    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;

    type Character is ...

end Standard;
```

Table 11-1 shows the default integer and floating-point types:

Table 11-1 Supported Integer and Floating-Point Types

Ada Type Name	Size
Integer	32 bits
Short_Short_Integer	8 bits
Short_Integer	16 bits
Float	32 bits
Long_Float	64 bits

Note that 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

```
package System is

    type Name is (Motorola_68k);

    System_Name : constant Name := Motorola_68k;

    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 := 2.0 ** (-31);
    Tick : constant := 1.0E-03;

    subtype Priority is Integer range 1 .. 254;

    type Address is private;

    Address_Zero : constant Address;


    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;

    function To_Address (X : Integer) return Address;

    function To_Integer (X : Address) return Integer;
private

    ...

end System;
```

Representation Clauses

The M68K/OS-9 cross-compiler supports the following representation clauses:

- Length clauses:

  —`for Access_Type'Storage_Size use X;`

  If X is static and equal to 0, no collection is allocated. Any attempt to evaluate an allocator will raise a storage error. (Other values of X, which need not be static, are honored.)

  —`for Task_Object'Storage_Size use X;`

  —`for Task_Type'Storage_Size use X;`

  —`for Fixed_Type'Small use X;`

- Record representation clauses: The compiler supports both full and partial representation clauses for both discriminated and undiscriminated records.
- Enumeration representation clauses.
- Address clauses for interrupts.

Restrictions on Array and Record Packing and Record Representation Clauses

- Arrays: Packed arrays of discretes (Integer and Enumeration types, including Booleans) are supported. Components of packed arrays occupy the minimum possible number of bits, which may range from 1 to 24.
- Records: A record field can consist of any number of bits between 1 and 32, inclusive; otherwise, it must be an integral number of words.
- Change of representation: Change of representation is supported wherever it is implied by support for representation specifications. In particular, implicit or explicit 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.

  The following example shows support for a change of representation of an array:

  ```
  type Arr is array(1..10) of boolean;
  type Brr is new Arr;
  pragma Pack (Brr)

  X : Arr := (1..10 => false);
  Y : Brr := Brr (X);
  ```

  Change of representation occurs in the type conversion to Brr.

Names Denoting Implementation-Dependent Components

There are no user-visible implementation names.

Interpretation of Expressions That Appear in Address Clauses

Address clauses can be used with statically allocated objects.

Unchecked Conversion

The target type of an unchecked conversion cannot be an unconstrained array type or an unconstrained discriminated type.

Machine Code

Machine-code insertions are not supported at this time.

Appendix A
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 B
Rational M68K/OS-9 Run-Time Error Messages

The following is a list of the M68K/OS-9 run-time error messages.

```
128:001 (***) Exception elaborating ADA runtime
```

An exception was raised and not handled while elaborating the Ada run time.

Please report this problem to Rational.

```
128:002 (***) Exception finalizing ADA runtime
```

An exception was raised and not handled while finalizing the Ada run time.

Please report this problem to Rational.

```
128:003 (***) Unhandled trap in main program
```

A machine trap occurred while executing the main program, for which no trap handler was installed.

Please report this problem to Rational.

```
128:004 (***) Unhandled trap in ADA task
```

A machine trap occurred while executing a task, for which no trap handler was installed.

Please report this problem to Rational.

```
128:005 (***) Exception elaborating library units
```

An exception was raised and not handled while elaborating a library unit, before executing the main program. The debugger can help identify where the exception occurred.

```
128:006 (***) Unhandled exception in main program
```

An exception was raised and not handled while executing the main program. The debugger can help identify where the exception occurred.

```
128:007 (***) Bad Entry Number
```

Please report this problem to Rational.

```
128:008 (***) <UNUSED ERROR CODE>
```

Please report this problem to Rational.

```
128:009 (***) Bad Open Alternatives
```

Please report this problem to Rational.

```
128:010 (***) Unexpected Reply
```

Please report this problem to Rational.

```
128:011 (***) Bad Message Id From Send
```

Please report this problem to Rational.

```
128:012 (***) Bad Status From Wait_Nonblocking
```

Please report this problem to Rational.

```
128:013 (***) Bad Status From Wait
```

Please report this problem to Rational.

```
128:014 (***) Bad Status From Length
```

Please report this problem to Rational.

```
128:015 (***) Bad Status From Remove_Message
```

Please report this problem to Rational.

```
128:016 (***) Bad Status From Send
```

Please report this problem to Rational.

```
128:017 (***) Bad Status From Send_Without_Priority
```

Please report this problem to Rational.

```
128:018 (***) Bad Status From Retrieve_Message
```

Please report this problem to Rational.

```
128:019 (***) Bad Status From Create
```

Please report this problem to Rational.

```
128:020 (***) Bad Status From Delete
```

Please report this problem to Rational.

```
128:021 (***) Bad Status From Delete_If_Empty
```

Please report this problem to Rational.

```
128:022 (***) Bad Status From Start_Timer
```

Please report this problem to Rational.

```
128:023 (***) Bad Status From Stop_Timer
```

Please report this problem to Rational.

```
128:024 (***) Bad Status From Set_Priority
```

Please report this problem to Rational.

```
128:025 (***) Bad Status From Fork
```

Please report this problem to Rational.

```
128:026 (***) Unexpected Null Task Id
```

Please report this problem to Rational.

```
128:027 (***) Bad Tcb Checksum
```

Please report this problem to Rational.

```
128:028 (***) Unexpected Null Layer
```

Please report this problem to Rational.

```
128:029 (***) Layer State Inconsistency
```

Please report this problem to Rational.

```
128:030 (***) Task Activated Twice
```

Please report this problem to Rational.

```
128:031 (***) Bad Message Size From Retrieve
```

Please report this problem to Rational.

```
128:032 (***) Bad Index From Wait
```

Please report this problem to Rational.

```
128:033 (***) Suspension State Inconsistency
```

Please report this problem to Rational.

```
128:034 (***) Bad Message In Queue
```

Please report this problem to Rational.

```
128:035 (***) Unemptiable Queue
```

Please report this problem to Rational.

```
128:036 (***) Reply Queue Not Empty
```

Please report this problem to Rational.

```
128:037 (***) Negative Delay Amount
```

Please report this problem to Rational.

```
128:038 (***) Bad Id from Start_Timer
```

Please report this problem to Rational.

```
128:039 (***) Entry Queue Not Already Deleted
```

Please report this problem to Rational.

```
128:040 (***) Reply Queue Already Deleted
```

Please report this problem to Rational.

```
128:041 (***) Acceptor Not Done Activating
```

Please report this problem to Rational.

```
128:042 (***) Parent Notified Twice
```

Please report this problem to Rational.

```
129:001 (---) Unhandled exception in Ada task
```

An exception was raised and not handled while executing a task. The debugger can help identify where the exception occurred.

```
129:002 (---) Unable to allocate the heap for the program
```

Allocation of memory to use for the program heap failed; the operating system call (F$SRqMem) returned an error. This may cause Storage_Error to be raised.

```
129:003 (---) Heap for the program has been exhausted
```

All memory allocated for use as the program heap has been used. This may cause Storage_Error to be raised. Increasing the heap size for this program may be appropriate.

Appendix C
M68K/OS-9 Directory Structure

This appendix indicates where important pieces of the Cross-Development Facility can be found. When the notation [current_load_view], [current_spec_view], or [current_code_view] appears, see the current release note to find out what to insert.

- 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.[current_spec_view].Units.M68k
  ```

- Run times and default linker-command files

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

  ```
  !Targets.Implementation.Motorola_68k.[current_code_view].
                                                  Units.Runtimes
  ```

- Object-module converter

  The procedure to invoke the object-module converter can be found in:

  ```
  !Targets.Implementation.Object_Conversion.[current_spec_view].Units
  ```

- File-transfer utilities

  The procedures to invoke the file-transfer utilities can be found in:

  ```
  !Targets.Implementation.Motorola_68k_Transfer.[current_spec_view].
                                                           Units.Os9
  ```

- Debugger

  The procedure for invoking the debugger can be found in:

  ```
  !Targets.Implementation.Motorola_68k_Debuggers.[current_spec_view].
                                                               Units
  ```

- OS-9 I/O packages

  Packages Os9_Io and Simple_Io can be found in:

  ```
  !Targets.Motorola_68k.Target_Interface
  ```

- Predefined I/O packages

  Packages Text_Io and Io_Exceptions can be found in:

  ```
  !Targets.Motorola_68k.Io
  ```

- Other predefined packages

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

  ```
  !Targets.Motorola_68k.Lrm
  ```

Index