Rational Compilation Integrator

User's Manual

RATIONAL

Product Number: 4000-00500

Rev. 2.0, December 1992 (Software Release 2_0)

This document is subject to change without notice.

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

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

Preface

The *Rational Compilation Integrator User's Manual* presents information about the Rational Compilation Integrator™ (RCI), a software product that can be added to the basic Rational Environment™.

The RCI enables you to use the Rational Environment to develop application programs that are compiled and executed on other platforms. The RCI integrates the Rational Environment's compilation and library systems with the following components of the remote compilation environment:

- The third-party vendor compilation system running on the remote platform
- The library system provided by the operating system on that remote platform

The RCI allows application developers to perform the following:

- Take advantage of the Rational Environment facilities for editing, semantic checking, library management, and configuration management
- Control remote compilation from the Rational Environment
- Manage consistency between the Rational Environment library system and the library system on the remote compilation platform

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

This manual provides instructions for using an RCI extension, the product that results from creating a customized RCI.

CONVENTIONS USED IN THIS MANUAL

The names of objects and the procedures for using the RCI are determined by the customization extension of the RCI and by the remote operating system for which the extension is built. Therefore, this manual can provide only examples of what named objects might look like and of what procedures might be followed.

Throughout this manual, the term *Custom_Key* refers to the target key named by the customizer; the term *Custom_* refers to the name of a particular extension, usually the same as the *Custom_Key*.

Space is provided in Appendix C, "Extension Tables," for the customizer to fill in valid values. This manual references these tables in appropriate places in the text.

AUDIENCE AND ORGANIZATION

The following subsections describe the audience and the sections of this manual that are addressed to those users. All users may find helpful information in Appendix D, "Quick Reference for Parameter-Value Conventions."

System Administrator

If you are responsible for setting up and maintaining hardware and software, you should read:

- Chapter 1, "Key Concepts": Introduces the concept of remote compilation. Read the "Setting Up For Remote Compilation: An Overview" section and the "Software Components: An Overview" section.
- Chapter 2, "RCI Setup Operations": Describes the steps for setting up the RCI environment. Of particular interest are the "Verifying the RCI Installation," "Enabling Remote Extensions Management," and "Setting Up Remote Communications," sections. The rest of the chapter deals with setting up library structures, which can be performed by the user.
- Appendix A, "Location of Components": Lists the location of important components of RCI software.

First-Time RCI User

If you are unfamiliar with the RCI, or if you want to refresh your memory on using the RCI, you should read:

- Chapter 1, "Key Concepts": Introduces the concept of remote compilation, draws parallels between the traditional development cycle and the native R1000 development cycle, introduces the integrated development cycle, outlines the software organization of the RCI, defines terminology, and gives overviews of setting up and developing in the RCI environment.
- Chapter 2, "RCI Setup Operations": Describes the steps for setting up the RCI environment. These are operations that need to be done only once or occasionally. Some sections are of most interest to a system manager or administrator; the sections "Setting Up Model Worlds," "Preparing To Set Up Library Structures," and "Setting Up Library Structures," contain information useful to users even if the user does not actually create library structures.
- Chapter 3, "Getting Started": Assists with the verification of RCI setup, explains the differences between R1000 and remote compilation, steps through the remote compilation process from creating code to executing it on the target, and explains what happens during that process.

Standard RCI User

If you have progressed beyond the first-time user stage, you may be interested in the following chapters:

- Chapter 4, "Using Batch Processing with the RCI": Describes using batch operation with the RCI.
- Chapter 5, "Maintaining File Consistency": Describes how consistency is maintained between units in different views on the host and between units on the host and remote machine.
- Chapter 6, "Library Management": Describes how joined views are kept consistent on the host and explains how the RCI allows you to maintain consistency between host views and remote libraries.
- Chapter 7, "Using Non-Ada Code with the RCI": Describes how to create, use, and maintain units in languages other than Ada under the control of the RCI.
- Chapter 8, "Package Rci": Provides conceptual information about the commands in package Rci and an alphabetic listing and description for each command.
- Chapter 9, "Package Rci_Cmvc": Provides conceptual information about the commands in package Rci_Cmvc as well as an alphabetic listing and description for each command.
- Chapter 10, "Package Cmvc": Provides conceptual information about the commands in package Cmvc as well as an alphabetic listing and description for each command that has specific RCI extended functionality.
- Appendix B, "Command Summary": Provides command syntax for commands in packages Rci and Rci_Cmvc.

RCI Customizer

Appendix C, "Extension Tables," provides blank tables as a convenient way of providing extension information to the end user. Fill in these tables with the appropriate information as you work through the customization procedure.

RATIONAL CUSTOMER SUPPORT

Rational customer support provides technical assistance by telephone, fax, fax on demand, and mail (including electronic mail).

Telephone and Fax

Telephone support is available Monday through Friday (except holidays) from 6:00 A.M. to 6:00 P.M. Pacific time in the United States. International telephone-support hours span the normal local business hours.

Sometimes Rational customer-support engineers will ask you to fax information to help them diagnose problems. You can also fax questions when that is more convenient for you than using the telephone. Please mark faxes "Attention: Customer Support" and be sure to include your return address and telephone number.

The telephone and fax numbers for Rational customer support are listed in the following table. The numbers include the country code, shown with a plus sign, and the area code, shown in parentheses. You do not need to dial these codes for local calls.

Country	Telephone Number	Fax Number
France	+33 (1) 47-17-41-77	+33 (1) 47-17-41-55
Germany	+49 (89) 797-021	+49 (89) 799-343
Sweden	+46 (8) 761-0600	+46 (8) 760-0026
Taiwan	+886 (2) 720-1938	+886 (2) 723-3899
United Kingdom	+44 (962) 877144	+44 (962) 870705
United States	+1 (800) 433-5444	+1 (408) 496-3636

Mail

You can get technical assistance on Rational products by sending electronic mail to support@rational.com. Electronic mail is acknowledged within one working day of its arrival at Rational.

You can also correspond with Rational at the following mailing addresses. Please mark correspondence to expedite its routing once it reaches Rational—for example: "Attention: John Smith" or "Attention: Customer Support."

Rational Mailing Addresses	
Rational International 1 Porchester St. Campbelltown SA 5074 AUSTRALIA	**Rational Scandinavia AB** Veddestavägen 24 S–175 62 Järfälla SWEDEN
Rational SARL Immeuble Delalande 16, rue Henri Régnault La Défense 6 F–92411 Courbevoie Cedex FRANCE	**Rational International** 3F–5, Jardine Fleming Bldg. 547, Kuang Fu S. Road Taipei, Taiwan R. O. C.
Rational GmbH Rosenstrasse 7 Grosshesselohe D–8023 Pullach im Isartal GERMANY	**Rational Technology, Ltd.** 28 Temple Street Brighton E. Sussex BN1 3BH U. K.
Rikei Corporation Shinjuku Nomura Bldg. 1–26–2 Nishi-Shinjuku Shinjuku-ku, Tokyo 163 JAPAN	**Rational** 3320 Scott Blvd. Santa Clara, CA 95054–3197 U. S. A.

QUESTIONS AND COMMENTS

If you have questions about using your Rational products, contact your Rational representative. If you would like to make comments about the usefulness or contents of this manual, use the forms at the end of the manual and send these to the Rational Publications Department, 3320 Scott Boulevard, Santa Clara, CA 95054-3197.

Contents

3 GETTING STARTED 43

4 USING BATCH PROCESSING WITH THE RCI 63

7 USING NON-ADA CODE WITH THE RCI 99

8 PACKAGE RCI 107

1 Key Concepts

This chapter introduces the concept of remote compilation, draws parallels among traditional, native Rational Environment™, and integrated development cycles, outlines the software organization of the Rational Compilation Integrator™ (RCI), defines terminology, and provides overviews of setting up and developing in the RCI environment.

Specifically, this chapter provides the following sections:

- Overview of the RCI
- Comparing development cycles
- Setting up for remote compilation: an overview
- Comparing Rational Environment and RCI compilation features
- Using the RCI: an overview
- Software components: an overview
- Terminology

OVERVIEW OF THE RCI

Integrated development is the process of designing, generating, and maintaining software on one machine (the *development platform* or *host*) that is compiled on another machine (the *remote compilation platform* or *remote machine*) under the control of the host for eventual execution on a predetermined machine (the *target*). The *target* can be the same machine as the remote machine (with a native compiler) or a different machine for which the code was compiled (with a cross-compiler). Integrated development uses the process of *remote compilation* to control the target compiler and linker on the remote compilation platform. This manual refers to the platform where compilation takes place as the remote compilation platform when discussing compilation activities and as the remote machine when discussing network operations such as creating remote directories and downloading files.

Depending on the target compiler, this remote compilation process may need to be handled in *interactive mode*, unit by unit, across the network, or in *batch mode*, where all units are transferred in at the same time and a minimal number of calls are made to the target compiler to process them.

To support this process, some sort of *cross-system consistency management* is also necessary to ensure that library structures and code are consistent between the host and remote compilation platform.

The RCI provides utilities to perform many aspects of integrated development automatically.

Integrated development can be very effective—and sometimes necessary. Several cases lend themselves to this approach:

- In some projects, software must run on a variety of target-hardware architectures and operating systems. Developing software separately (on different host environments) for each target could result in different program characteristics for each target. Using one development host is an effective way to organize development, standardize programs, provide a common user interface, and reduce duplication of effort.
- The host may be more effective than the target for producing well-engineered code on a tight schedule for large projects. The host's advantage may come from its specialization in development tasks or from limitations of the target—or both.
- The target may be unavailable during part of the project. The ability to develop and test code on the host allows the project to proceed in a timely manner.

Rational's RCI products provide the means to develop programs on the R1000®, Rational's software-engineering server, for a variety of targets, taking advantage of the R1000's specially designed software-engineering support environment. A large portion of a project's software can be developed and tested on the Environment. Target-dependent software can be developed and integrated on the Environment and tested on the target. The RCI's integrated approach is shown in Figure 1-1.

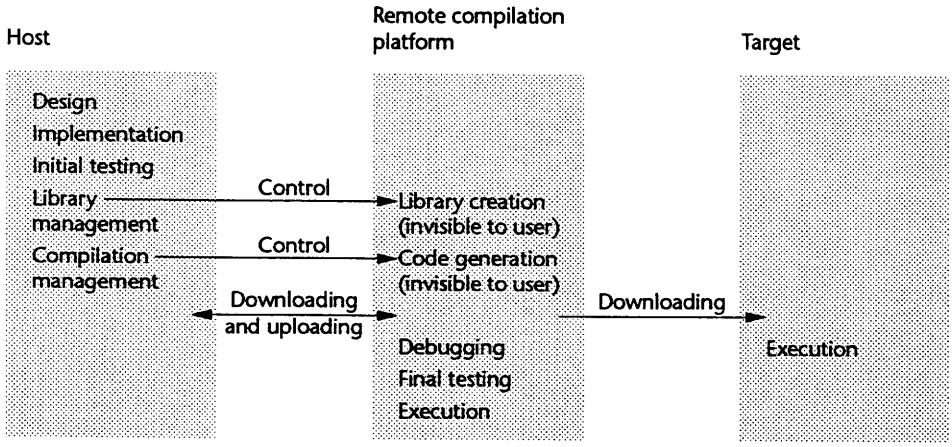

Figure 1-1 RCI Integrated-Development Approach

For source code that is either target-independent (portable) or target-dependent, modifications made on both the Rational Environment development host and the remote compilation platform can be managed efficiently for different targets.

COMPARING DEVELOPMENT CYCLES

A *development cycle* is a repeated series of operations within the software-development process.

- A *traditional cycle* might include compiling, assembling, linking, loading, executing, and debugging, all occurring on one machine.
- When code is developed entirely on the Rational Environment, the operations differ enough from a traditional cycle to be called the *native R1000 development cycle.*
- When code is developed on the Rational Environment host for execution on a different machine, the compilation processes on both host and remote machine make up the *integrated development cycle.*

These cycles are compared in the following subsections.

Traditional and Native R1000 Development Cycles

Depending on the tools you use, one operation may perform the equivalent of several operations using different tools. For example, syntactic and semantic checking can be performed as separate actions in the Rational Environment but may occur during compilation on another system. As another example, coding an Ada unit in the Environment may include both assembling and linking, which might require separate actions on another system.

Figure 1-2 illustrates a development cycle in terms of traditional operations and Environment actions. There is no one-to-one correspondence between the operations of the two development cycles; for example, the Environment steps labeled "check syntax" through "code" cover the same underlying operations as compile, assemble, and link in the traditional cycle.

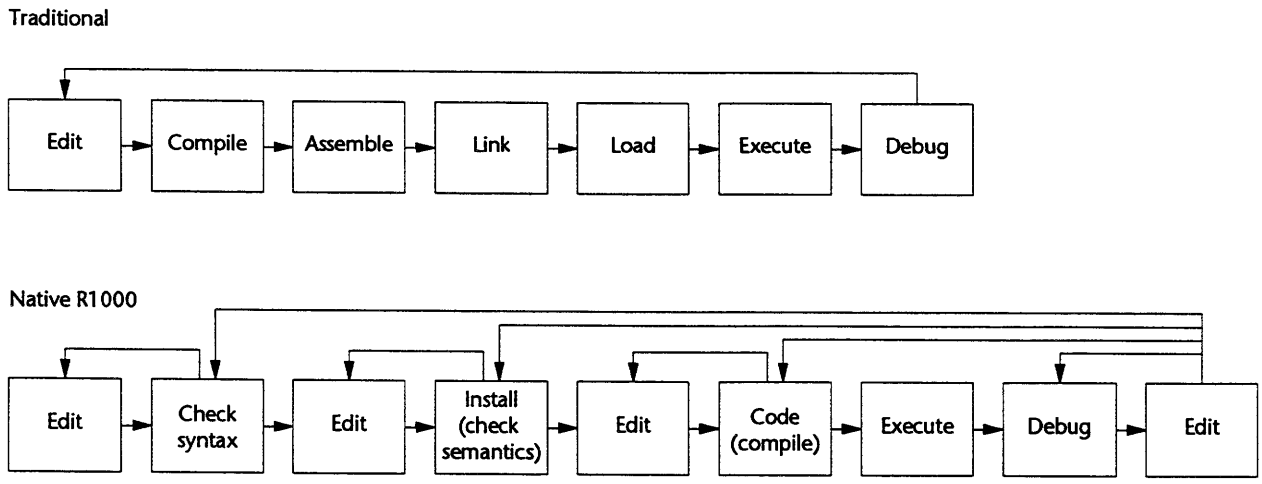

Figure 1-2 Traditional and R1000 Native Development Cycles (User's Perspective)

The power of the native cycle lies in the user's ability to return to the edit operation and to perform the operations incrementally. For additional information on the native R1000 development cycle, see the *Rational Environment Reference Manual.*

Integrated Development Cycle

In an integrated development cycle, some of the operations described above or parts of them are carried out on the host in the same manner as for the native cycle, some on the host with changes based on the intended target, and some on the remote machine itself. In integrated development, the cycle includes additional communication operations: *downloading* and *uploading* (transferring code from the host development platform to the remote compilation platform and vice-versa), host-remote communication during the operations of *remote compilation* and *remote linking,* and the transfer between host and remote machine of cross-system consistency-management information. The RCI automates these steps.

The recommended development cycle using the RCI begins with the *native R1000 development cycle*—that is, creating and testing code on the Rational Environment

for target-independent operation. Then, when you are ready to test your code using the target compiler on the remote compilation platform, the *integrated development cycle* begins. You can repeat this process as you iteratively test your system. From the user's view, the integrated development cycle is very similar to the native cycle, as shown in Figure 1-3.

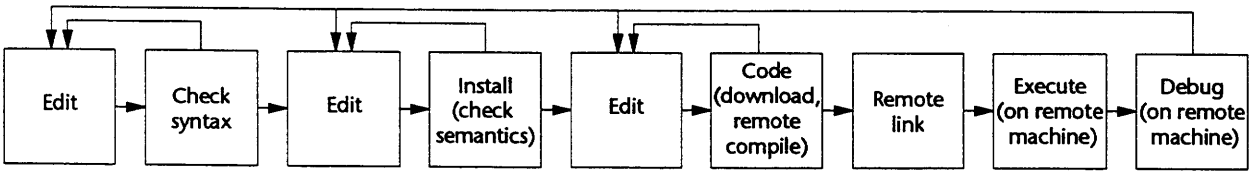

Figure 1-3 Integrated Development Cycle (User's Perspective)

The relationship of various steps to the location in which they execute is discussed in the next subsection. Applying the integrated development cycle is discussed further in Chapter 3, "Getting Started."

Examples of Software Development Cycles

The development cycles for the Rational Environment and for the RCI are shown in Figure 1-4 so that you can compare the processes.

All operations in the native R1000 development cycle take place on the host, as shown in the simplified portion of Figure 1-4 labeled "R1000: Native cycle."

With the RCI, a native development cycle may first take place entirely on the host. Then, as shown in Figure 1-4, the integrated development cycle begins, in which compilation and linking take place on the remote compilation platform but are controlled from the development host. Finally, execution and debugging take place entirely on the remote machine. In addition, some of the features of configuration management and version control *(CMVC)* are extended to maintain consistency between the development and compilation platform for source code, object code, and library structures.

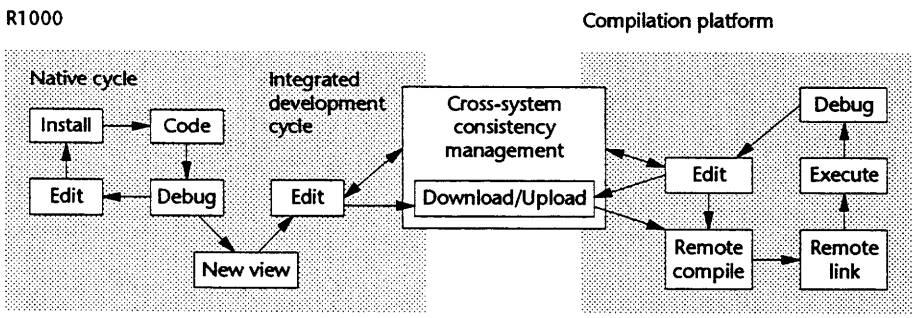

Figure 1-4 Integrated Development Cycle for RCI

SETTING UP FOR REMOTE COMPILATION: AN OVERVIEW

Before you can use the RCI, the appropriate software components must be in place and several switches and files must be set properly. The system administrator may be responsible for some of the setup operations and the user for others. This section provides an overview of:

- Setting up the network and hardware
- Setting up the remote environment
- Setting up the Rational Environment

Figure 1-5 shows the location and relationship of components and library structures discussed in the following subsections.

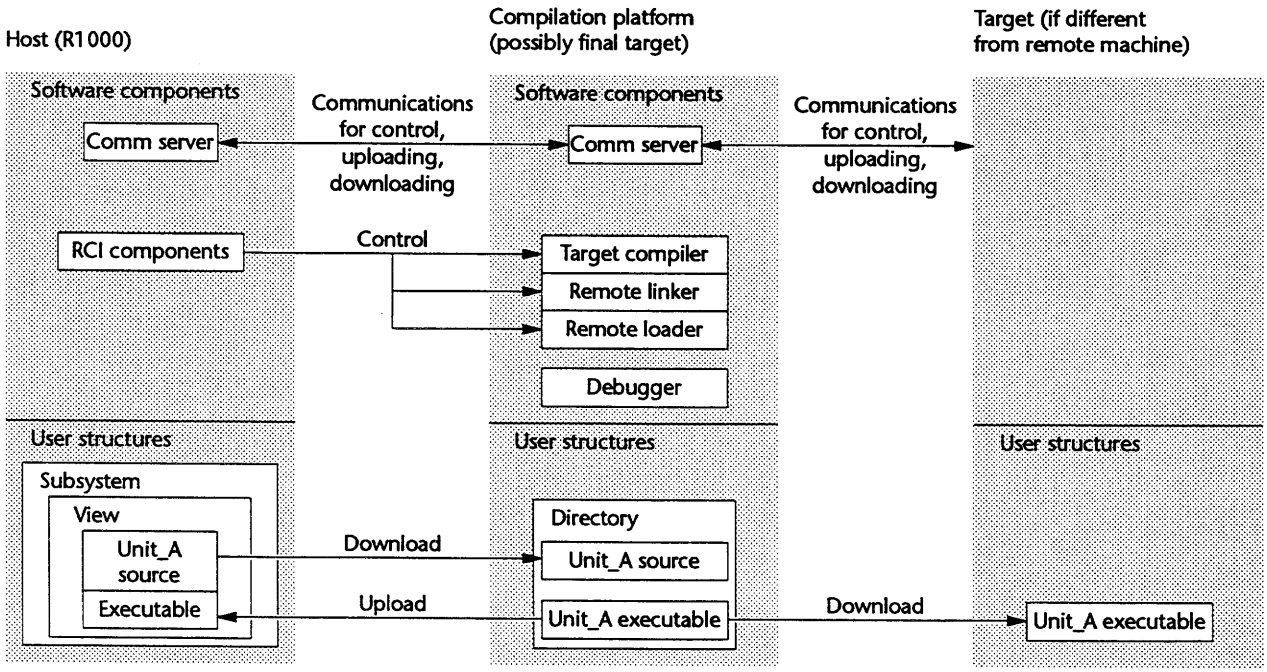

Figure 1-5 Location of Components and Library Structures

Setting Up the Network and Hardware

To use the RCI, the development host and the remote compilation platform must be connected to the same network, and each machine must be provided with information on how to find each other. For the Rational Environment to communicate with the remote machine, the *transport name map* must be correctly set on the R1000 to include the correct network name and address of the remote machine. This information is described in more detail in the "Setting Up Remote Communications" section in Chapter 2. Also refer to the File Transfer Protocol (FTP) book of the *Rational Networking—TCP/IP Reference Manual* for additional information.

Setting Up the Remote Compilation Environment

The RCI uses either DTIA or Telnet/FTP, selected in the customization template, to communicate with the remote compilation platform. Communication requires network servers operating on the development host and remote compilation platform.

Telnet/FTP customizations do not need the DTIA package.

For customizations that use DTIA for network communication, the Rational DTIA server must be installed and running on the remote machine before remote compi-

lation can take place. The RCI automatically loads and runs the DTIA remote-operations package for DTIA extensions.

For additional information, see the RCI installation notes and the "Verifying the RCI Installation" subsection in Chapter 2.

Setting Up the Rational Environment

Setup in the Rational Environment involves making sure that the RCI software components exist and are running on the R1000 and establishing library structures for RCI development.

Software Components

The various levels of software required to develop software under the RCI are set up during the RCI installation process. These components are described in "Software Components: An Overview," later in this chapter. Verifying their installation is described in the "Verifying the RCI Installation" subsection in Chapter 2.

Library Structures

Development for the native R1000 and integrated development cycles takes place in library structures (subsystems, paths, views, directories). For the RCI, *subsystems* are the highest level of R1000 library structure, and development takes place in *combined views* within the subsystems. Subsystems for the RCI are discussed in detail in the "Preparing To Set Up Library Structures" and "Setting Up Library Structures" sections in Chapter 2.

When creating the R1000 library structures, you are saved from having to define every aspect of the library by copying a *model world* that has already been defined with defaults appropriate to the RCI. A model world provides:

- A *target key* that identifies the target for which code is to be developed and determines what compilation system to use
- A library-switch file that defines how the RCI operates
- A set of external links that give visibility to predefined Ada units that reside outside the subsystem

Model worlds and their components are described in more detail in the "Setting Up Model Worlds" section in Chapter 2.

Under the RCI, software is often developed in two roughly parallel libraries: one for the native R1000 environment and one on the R1000 that targets the remote environment. The program library on the remote machine is associated with the R1000 remote-targeted library. Although you can make changes in the remote library and then transfer them back to the host, this practice is not recommended during development.

For example, if you were to develop an Ada main unit called Unit_A, you might first create it and test it using Rational Environment utilities in a view with an R1000 target key (an *R1000 view*). Later, you would use CMVC tools to copy the file into a view with a *Custom_Key* target key (an *RCI view*), which would also create a copy of the file in a directory on the remote machine. With the RCI cross-system consistency-management tools, you could control consistency between the R1000 and remote libraries.

COMPARING FEATURES OF RATIONAL ENVIRONMENT AND RCI COMPILATION

The integrated RCI compilation system is similar to the native R1000 compilation system from the user's perspective. For both, Ada units exist in an archived, source, installed, or coded state. For both, the Rational Environment manages all dependencies and guarantees that all required units are available in the proper state.

The same compiler commands are used to control both compilation systems. See the Editing Specific Types (EST) and Library Management (LM) books of the *Rational Environment Reference Manual* for more information.

Since processing for remote compilation involves more than one machine, it is useful to describe this processing in more detail. This is done in the "Remotely Compiling And Linking A Simple Ada Program" section in Chapter 3.

The following subsections describe the differences between the native R1000 compilation system and the RCI, citing references to further detail.

Semantic Checking

The RCI supports standard LRM Ada with some implementation-dependent features for the chosen target. Implementation-dependent differences in Ada source code input can be determined by comparing the implementation-dependent features given in Chapter 13 and Appendix F for the R1000 in the *Reference Manual for the Ada Programming Language* (Ada LRM) and those in the Appendix F of the Ada LRM for the target compiler. Some differences are discussed in the "Creating An Ada Program For Remote Compilation" section in Chapter 3.

Because of these features, semantic checking, which occurs when a unit is promoted to the installed state, differs slightly between the two compilers—for example, there are differences in the pragmas that are recognized. The user interface is the same. See the "Creating An Ada Program For Remote Compilation" and "What Happens During the Installing Step" sections in Chapter 3.

Pragmas and Other Implementation-Dependent Features

Pragmas and attributes allowed by semantic checking are determined by the target key. The definitions of packages System and Standard are also determined by the target key, as are the definitions of other predefined packages such as Calendar and Text_Io. The "Using Implementation-Dependent Pragmas" subsection in Chapter 3 lists the implementation-dependent pragmas, and Appendix A provides information about the location of predefined packages.

Generics and Inlined Subprograms

The R1000 architecture supports shared-code generics so that multiple instantiations of a generic share the same code. The target compiler may use macro expansion to implement instantiations of generics, so multiple instantiations might yield multiple copies of the code. This could introduce additional dependencies not present in R1000 compilation.

Inlining of a subprogram whose body is in a different compilation unit could introduce additional dependencies with the target compiler.

The target compiler may support inlining of subprogram calls to subprograms defined in other units, which is not supported by the R1000 compiler.

Because of these additional dependencies, the demotion of a generic body or an inlined body under the RCI automatically demotes all units containing instantiations or inlined calls to the same state as the demoted unit. This action depends on the scope of the command.

Packed Records and Arrays

For the R1000, all arrays and records are bit-packed by default because the storage unit for the R1000 is a bit. For other machines, the storage unit may be different. For some remote machines, for example, the storage unit is a byte and the default is byte-aligned. Minimization of storage must be requested explicitly with pragma Pack, a length clause, or a record representation specification. Further information can be found in Appendix F of the Ada LRM for the target compiler.

Record Representation

In the absence of record representation specifications, the R1000 and target compilers may lay out record fields differently. Using record representation specifications, record layouts for the RCI compiler can be controlled precisely. See the "Using Representation Clauses" subsection in Chapter 3 for further information.

Options and Switches

There are additional compiler-option switches for the RCI to control optional outputs and target-compiler activity. See the "Setting Session and Library Switches" subsection in Chapter 3.

Incremental Operations

Incremental operations are possible in both native R1000 and RCI views, but there are some differences in when they can be used.

In an R1000 view, coded package specifications can be changed incrementally. In RCI views, incremental operations on coded objects are prohibited. The user can perform incremental operations on units in the installed state in an RCI view, as in an R1000 view.

Table 1-1 shows the object state and the incremental operations that can be performed in that state. For more explanation about performing incremental operations, consult the *Rational Environment Reference Manual.*

Table 1-1 Incremental Operations

Object State	Incremental Operation	R1000	Custom_Key
Installed	Add, change, or delete a statement, declaration, or comment	Yes	Yes
Coded	Add, change, or delete a comment anywhere or a declaration in a library-unit package specification	Yes	No

Executable and Object Modules

For the native R1000 compilation system, any procedure in the coded state is executable. The RCI compiles Ada programs for other processors; an executable module is produced only when an Ada main unit is compiled and linked remotely.

Output

On the output side of the two compilation systems, the major result is the same: an executable module. For the RCI, there are additional outputs and differences in output that result directly from the different customization options and implementation-dependent features.

The target compiler and linker produce files that can be retrieved from the remote compilation platform and moved onto the host as files associated with the coded unit. These *associated files*, such as the object module, source listing, and assembly-language listing, are discussed in the "Output from the RCI Compiler and Linker" subsection in Chapter 3.

Command Windows

A command in a command window attached to an RCI world or view runs an R1000 program in the same way that it does in any command window.

Simultaneous Compilations

Promotion and demotion operations that occur simultaneously in the same view may be restricted due to the nature of dependency management, as described in the "Rci State Information" section in Chapter 6.

In addition, the RCI that you are using may establish a set of *tokens* that limit the number of simultaneous user operations. See the RCI release notes for further information.

USING THE RCI: AN OVERVIEW

After appropriate setup steps have been performed, you can begin RCI development by creating and testing code on the host (the native development cycle) and then move to the process of integrated development.

Read the "Comparing Features Of Rational Environment And Rci Compilation" section before reading this section.

Overviews are presented in the following subsections for:

- Developing and executing code on the host
- Developing and executing code for the target machine

Developing and Executing Code on the Host

An optional way to begin remote compilation is to develop and test code as much as possible in the native R1000 environment; this is referred to as the *native R1000 development cycle.* Do this in using views with an R1000 target key.

Often, code for a project is developed in at least two subsystem views: one with an R1000 target key that contains code that is completely portable between the host and any intended targets (an *R1000 view*), and one with the custom target key that contains code intended for execution on the host or different targets (an *RCI view*).

The process of creating source code, checking semantics, installing, coding, executing, debugging code, and performing configuration management entirely on the R1000 is described in the *Rational Environment Reference Manual.*

Advantages of the R1000 Development Cycle

If you choose to begin with the recommended R1000 development cycle, it allows you to begin developing a project and testing code in the tightly controlled environment provided by Rational CMVC without concern for network connections, file transfers between host and remote machine, or ensuring consistency between host and remote machine.

Turnaround time for unit testing is much faster on the Rational host for a number of reasons, including incremental compilation, run-time evaluation of dependencies, and the absence of generic macro expansion.

Moving to the Integrated Development Cycle

After you are satisfied that the code has been verified as thoroughly as possible in the native R1000 environment, the code in all R1000 subsystem views is copied to and joined with RCI combined views (libraries with a *Custom_Key* target key) where downloading of the code to the remote machine, remote compilation and linking, and cross-system consistency management can occur. This copying of code is described in "Setting Up Library Structures" in Chapter 2 and "Consistency between Views on the Host" in Chapter 5. This begins the integrated development cycle, described in the next subsection. As you test, you can repeat this process.

Developing and Executing Code for the Target Machine

The steps in the *integrated development cycle* for the chosen target, the second phase of development, are very similar to those in the native R1000 cycle. The following steps are described in more detail in Chapter 3, "Getting Started."

- *Creating/modifying Ada source:* The facilities for producing Ada source are the same as for the native R1000 cycle. Typically, much of the code for the target is first developed in the native R1000 cycle and then ported to the integrated development cycle.
- *Checking syntax, checking semantics, and installing:* The actions done by the user are the same as for the native R1000 cycle; there is some difference, however, between how the native Rational Environment and RCI compilers do semantic checking, because there are target-dependent semantics.
- *Writing non-Ada subprograms (optional):* Units in languages other than Ada for the remote compilation platform can be written directly in an R1000 text file. The text file (called a *secondary*) can be associated with an Ada unit (called a *primary*), which controls when the text file is automatically downloaded and compiled or assembled on the remote machine. This is described in more detail in Chapter 7, "Using Non-Ada Code with the RCI."
- *Coding:* In interactive mode, promoting a unit to the coded state automatically downloads the Ada code to the remote machine if necessary, invokes the target compiler, and produces an object module on the remote machine. This module is ready to be linked. In batch mode, you create a batch script that controls the remote compilation by identifying what units to transfer and how to handle the target compiler. This compilation step can include automatic linking and executing the code. This process is discussed in Chapter 3, "Getting Started," and Chapter 4, "Using Batch Processing with the RCI."
- *Remote linking:* After the closure of a main unit is coded, the user can generate a target executable. The host user must invoke the remote link process, which creates an executable module from the compiled object modules. This is discussed further in Chapter 3, "Getting Started."
- *Initiating execution and debugging:* Execution and debugging of a program are initiated directly on the remote machine using remote operating-system utilities.
- *Maintaining consistency:* Cross-system consistency management consists of RCI extensions to the Rational CMVC facilities to maintain consistency between the host and remote machine for libraries, Ada source, and non-Ada source files. Some operations are performed automatically; some must be requested by the user. This is described further in Chapter 5, "Maintaining File Consistency," and Chapter 6, "Library Management."

SOFTWARE COMPONENTS: AN OVERVIEW

A remote compilation *component* is the software that carries out one or more operations under the control of the RCI. Remote compilation also uses other components, such as an editor, that are not particular to RCI operations.

General-Purpose Environment Software Components

The following facilities and components are not specific to the RCI but are required for various RCI operations:

- The CMVC facility: See the Project Management (PM) book of the *Rational Environment Reference Manual*, especially the "Using CDFs with Subsystems" chapter.
- The library system: See the Library Management (LM) and Session and Job Management (SJM) books of the *Rational Environment Reference Manual*.
- Model worlds, including library-switch files and target keys that refer to the RCI: See "Setting Up the Rational Environment," in this chapter.
- Network-communication mechanisms: These provide RPC (Remote Procedure Call) mechanisms for moving objects and executing commands between the host and remote machine. This includes servers running on both the host and the remote machine. For RCI these mechanisms include Telnet/FTP or DTIA, depending on your extension.

RCI-Specific Software Components

The following components are tools that are particular to the RCI and run on either the R1000 or the remote machine under the control of an RCI user:

- RCI compilation job
- RCI user interface: packages Rci and Rci_Cmvc
- Customization components
  - Templates
  - Predefined units
  - Extensions
  - Extensions job

Information on where to find these components is given in Appendix A.

RCI Compilation Job

The remote compilation job acts as a server through which all command and compilation operations of the RCI are routed. It runs on the host environment and makes the facilities of the RCI available to the host system. It coordinates the RCI compiler, interfaces, customization extensions, and communications.

The remote compilation job controls the RCI compiler. The RCI compiler runs on the R1000 host and is used in the installing and coding steps. It controls the downloading and compilation of sets of interdependent Ada units and their secondary non-Ada units. It invokes the target compilation system to produce object code.

The RCI compiler is similar to the native R1000 compiler from the user's perspective. Similarities and differences are discussed in "Comparing Features of Rational Environment and RCI Compilation," earlier in this chapter.

RCI User Interface

The RCI provides user commands through two interface packages, Rci and Rci_Cmvc. Chapter 8, "Package Rci," and Chapter 9, "Package Rci_Cmvc," describe these commands. In addition, the Rational Environment provides RCI support through package Cmvc. The RCI extensions to CMVC are described in Chapter 10, "Package Cmvc."

These packages contain commands to handle RCI user operations. These operations control RCI activities, including the following:

- Batch-compilation operations, which operate in batch mode rather than interactive mode, build batch scripts to control compilation on the remote compilation platform, transfer units either over the network or by tape, check the build state of units, transfer associated files back to the host, and transfer units to different machines. These operations are described in Chapter 4, "Using Batch Processing with the RCI."
- Cross-system consistency management involves tools, some automatic and some that must be invoked by the user, that ensure a consistent development environment between the host and the remote compilation platform. Source and object code, both Ada and non-Ada, can be maintained in a consistent state, as can libraries and imports. Information about the state of consistency, called *RCI state information*, is discussed in more detail in Chapter 5, "Maintaining File Consistency," and Chapter 6, "Library Management."

  The cross-system management tools are invoked from the R1000; there are no tools on the remote machine. These tools establish parallel units and libraries and detect state inconsistencies between the host and remote machine. Unlike CMVC, they do not prevent inconsistencies from occurring.
- Host and remote library management is handled through packages Cmvc (or Rci_Cmvc) and Rci, and operations enabled through library extensions in your customization. Rci and Cmvc (or the corresponding Rci_Cmvc) commands control library operations that involve imports and views.

  The Cmvc view-creation commands have corresponding Rci_Cmvc commands to allow you to directly specify the Remote_Machine and Remote_Directory parameters for remote libraries. The CMVC commands use a default RCI switch scheme to create those values.
- Non-Ada unit support is provided through management of text files containing source from other languages. The RCI uses the compilation dependencies of an Ada host-environment unit to determine processing of this *secondary text file* on the remote compilation platform.

Customization Components

The RCI includes software components to allow you to customize the facility for your specific platform. As an RCI user, you do not need to work with these components. Appendix C provides an area to record the specific details of your extension. For additional information, see the user's guide for your extension.

The customization components contain the following elements:

- The RCI *customization templates* provide a framework to define the characteristics of your target compiler, remote operating system, and network communica-

tions mechanism. The RCI uses this information for semantic checking, construction of remote commands and libraries, remote naming of files, and network communication protocols. Your customizer uses these templates to create an RCI environment for your specific target compilation system and platform.

- The *predefined library units*, such as Text_Io, are as specified in Annex C of the Ada LRM. The host environment needs access to LRM-defined packages for the target compilation system to use for semantic checking. Your customizer creates these packages on the host by entering the contents printed in the Ada LRM for your target compilation system.

  Caution: *It is illegal to copy or upload these packages from the target compilation system. Contact your compiler vendor for information about your license and restrictions.*

- The library management and compilation *extensions* allow the customizer to program additional operations involving remote library management (build, rebuild, and destroy remote libraries) and compilation (promote and demote).
- The *extensions job* enables library management and compilation extensions. This job is run automatically when the RCI compiler job runs.

Non-Rational-Supplied Components

The following components are invoked by the RCI but are not supplied by Rational. They are assumed to exist on the remote compilation platform as described in the RCI installation instructions:

- Remote compiler: This is the target compiler, which runs on the remote machine.
- Remote linker: This is the part of the target compilation system that links object modules.

TERMINOLOGY

Ada LRM: *Reference Manual for the Ada Programming Language.*

batch mode: The mode in which coding units on the host does not cause remote compilation to occur.

CMVC: Configuration management and version control across objects in the Rational Environment.

coding: The process of promoting a unit to the coded state in the Rational Environment, which downloads source code to the remote machine and invokes the target compiler to produce machine-dependent object code. Under batch mode, coding a unit simply causes the RCI to record the coding time in the state information for that unit.

compilation platform: The machine that hosts the compiler controlled by the RCI.

cross-system consistency management: Consistency management across objects in the Rational Environment and remote libraries.

Custom_Key: The compilation component of the target-key name as defined by the RCI customizer; for example: I186_Vms_Ddc.

downloading: The process of transferring an object from the host to the remote machine or to the target machine.

DTIA: Distributed tool-integration architecture, a Rational-supported network communication mechanism.

executable module: Code that has been compiled and linked and can be executed as a stand-alone application.

extension: The RCI development environment produced for a specific remote compilation platform and target. This is done through customizing.

FTP: File Transfer Protocol, a bidirectional mechanism for file transfer used to move text and binary data files between two or more machines.

host: The primary development and testing machine in a remote compilation environment, usually the R1000.

installing: Promoting a unit from the source state to the installed state, during which semantic checking takes place.

integrated development cycle: The process of using the RCI to create, manage, compile, and link code using both the host and remote machines to produce executable units for the target machine.

interactive mode: The mode in which coding units on the host causes the RCI to download and compile those units on the remote compilation platform.

joined: Files that represent the same code in two views on the host, where package Cmvc allows changes to only one copy at a time.

module: The file or set of files that, when compiled, produces a single object module.

native R1000 development cycle: The creation and testing of code in a Rational Environment library that contains an R1000 target key—that is, the application is intended to run on the R1000 rather than on the target.

object module: A binary file produced by a compiler or an assembler that contains code, data, and relocation information.

primary: An Ada unit on the host that is associated with one or more secondary text files; the primary is used to determine the compilation ordering of the secondaries but is not (usually) itself compiled.

program: The set of modules that, when linked, results in a single executable module.

R1000 view: A view with an R1000 target key.

RCI path/view: A view with a target key defined by the particular customization of the RCI (the *Custom_Key* target key).

remote compilation platform: The machine that hosts the compiler controlled by the RCI; same as compilation platform.

remote compilation system: The compilation system running on the remote compilation platform.

remote import list: A method on the remote machine of providing visibility to code outside a specific library (referred to in the Rational Environment as *links*). For some compilers, this is an editable text file known also as the *library list file.*

remote library: A library structure on the remote machine that often consists of a directory and its enclosed remote program library and remote import list.

remote linker: The linker on the remote machine that is controlled by the Rci.Link command.

remote machine: The machine that hosts the target compiler controlled by the RCI.

remote program library: A library on the remote machine containing executable modules.

secondary: A text file on the host that is associated with an Ada-unit primary; the text file (usually) contains source code that can be compiled or assembled on the remote machine but not on the host; the associated primary determines the method and order of the secondary's compilation.

target: The machine on which remotely developed code ultimately executes.

target compilation system: The compiler on the remote compilation platform (remote machine) that is controlled by RCI operations. The compiler produces executable modules for the target machine.

target-dependent: Commands or code that executes correctly only for a specific target or that depends on features of a given target.

target key: A piece of information associated with each view that determines the target-code generator and other operations that take place in the view. RCI target keys contain information included in the compilation component of the target-key name, including target architecture, remote operating system, and compiler vendor. A target key consists of two components: a compilation component, which provides compilation information, and a design component, which defines what program design language to use.

Telnet: A bidirectional network communications mechanism used to establish connections between machines.

uploading: The process of moving an object from the remote machine to the host for permanent storage on the host.

2 RCI Setup Operations

This chapter helps users to create their desired development environments and provides information for the system manager for starting and operating the RCI on the host and on the remote compilation platform.

Tasks for the system administrator and customizer:

- Running the RCI
- Starting the RCI
- Verifying the RCI installation
- Enabling remote extensions management
- Setting up remote communications mechanisms

Tasks for the user:

- Setting up remote communications and login information
- Setting up model worlds
- Preparing to set up library structures
- Setting up library structures

RUNNING THE RCI

Starting the RCI

After installation you can start the RCI using the following methods:

- At machine initialization: The RCI is automatically started as a part of the boot process with machine initialization. The process starts the Rci_Compiler RevX_X_X job (your current release) and registers selected target keys. The selected keys are listed in one of the following files:
  - !Machine.Initialization.Site.Rci_Configuration
  - !Machine.Initialization.Local.Rci_Configuration

  The site file contains targets that apply across all systems at a particular site. The local file contains targets for a single machine only.

  The RCI machine-initialization routine allows you to specify activities in the two Rci_Configuration files located in !Machine.Initialization.Site and !Machine.Initialization.Local. The first field specifies the target-key name. An optional second field on each line in the Rci_Configuration files identifies the activity. The format is as follows:

  ```
  <TARGET KEY NAME> <ACTIVITY>
  ```

The <TARGET KEY NAME> specifies the full pathname of the customization subsystem to register at boot time. If you do not provide a full pathname, initialization uses the default context, !Targets.Implementation.Rci_Customization.

The <ACTIVITY> specifies the activity used to find an entry for the customization subsystem. If you do not provide this field, it defaults to !Machine.Release.Current.Activity.

Using this optional field eliminates the requirement for customers to update !Machine.Release.Current.Activity for site-specific customizations.

- During normal operations: Run Start_Rci_Main to start the Rci_Compiler RevX_X_X job. Then use *Custom_Key*.Register to register each customization and run the associated extensions job.

Verifying the RCI Installation

To verify that the Rational Environment components of the RCI are running, use this command on the R1000:

```
What.Users
```

The following jobs, or jobs with similar names as described in the RCI release notes, should be running:

- Communications Server for host and remote machines
- Rci_Compiler RevX_X_X
- Extension job, either FTP or DTIA in either interactive or batch mode
  - Rci_RevX_X_X_*Custom_Key*_FTP (for interactive Telnet/FTP extensions) or Rci_RevX_X_X_*Custom_Key*_FTP_Batch (for batch mode)
  - Rci_RevX_X_X_*Custom_Key*_DTIA (for interactive DTIA extensions) or Rci_RevX_X_X_*Custom_Key*_DTIA_Batch (for batch mode)

See your current release notes for more information.

To verify that the remote components of the RCI are running, use the appropriate command from the remote operating system to display running processes. Refer to the installation instructions and release notes for further information on what processes must be running.

Killing the RCI

To kill the RCI, run Kill_Rci_Main. This command kills the Rci_Compiler job, unregisters all extensions, and kills any running extensions jobs.

Enabling Remote Extensions Management

The RCI provides mechanisms in each customization extension to extend operations involved with library management and compilation.

The RCI maps Rational host environment (RCI) views to remote libraries. CMVC library-management operations (Make_Path, Import, Remove_Import, and so on) can be extended to automatically perform similar simultaneous operations on the library system for the remote operating system so that users of the RCI do not have

to manage remote libraries as a separate task from managing host libraries. These library extensions are not included in all customizations.

If these extensions are not enabled, the RCI generates warnings during operations, and users must verify that host library structures are consistent with remote library structures.

The RCI also allows compilation-related operations, such as promote and demote, to be extended. Your customizer provides those extensions as they apply to your environment.

The special extension job starts automatically after the RCI is running. This job is initiated with the Register procedure from the !Targets.Implementation.Rci_Customization subsystem, which can be run during machine initialization or anytime thereafter. The extensions are automatically run by the Register procedure. You may need to register your extension if in the process of troubleshooting, you have killed all running RCI jobs, or your customizer has made a change to the customization templates for your extension. See "Verifying the RCI Installation," above, and the current release notes for additional information.

To register your extension:

1. Enter the *Custom_Key*.Register command and press [Complete]:

```
Custom_Key.Register
(Batch_Mode             :Boolean = "False",
 Allow_Standard_Rebuild :Boolean = "False";
```

2. Complete the following parameters:
   - Batch_Mode

     Specifies that the extension should be registered in batch mode. In this mode, the RCI supports creation of batch scripts but does not automatically download units and compile them on the remote platform as it does in interactive mode.
   - Allow_Standard_Rebuild

     Indicates whether the RCI should automatically rebuild a new package Standard, if the Standard_Version defined in Get_Predefined_Info is different from the current Standard_Version in the predefined world. Changing the Standard_Version requires demotion of package Standard and all dependent units. Unless specified in the customization, the default value of Allow_Standard_Rebuild in RCI_Customization.Register is False.
3. Press [Promote] to register your extension.

SETTING UP REMOTE COMMUNICATIONS

The host and remote machines must be connected to the same network to use the RCI. In addition, the Rational Environment must be informed of the existence and attributes of the remote machine for all users and of specific login information for each remote user.

Enabling Remote Access for All R1000 Users

The !Machine.Transport_Name_Map file (the *transport name map)* on the R1000 must contain an entry for the chosen remote machine. This file provides a mapping

between machine names and Internet addresses. The RCI needs this information to transfer Ada units and to execute remote operations.

In the following example, which shows one line of a Transport_Name_Map file, the communication protocol is TCP/IP, the chosen target is identified both by the name Tilden and by its Internet address (89.64.128.6), and Tilden is running a UNIX®-based operating system:

```
tcp/ip 89.64.128.6 Tilden Unix
```

You may need to set up similar files for communication on the remote machine. Many systems require a file similar to the Transport_Name_Map to identify other machines on the network. UNIX systems use .rhost and .rlogin files for connection and security. Refer to your release notes and user's guide for the remote operating system for information about setup requirements on the remote compilation platform.

Enabling Telnet Ports for RCI

When you use Telnet communication with remote compilation platforms, the R1000 host uses free Telnet ports for communication with the remote machine. This requires that one or more Telnet ports are reserved for RCI use—that is, they are not enabled for login. System administrators should reserve a number of Telnet ports equal to the number of RCI users. Use Operator.Disable_Terminal to control the ports.

The RCI caches remote connections to improve throughput during interactive RCI operations. After a user establishes a connection, the RCI does not acquire new connections for the same user to the same remote machine, for as long as the original connection remains in the cache.

The RCI caches a maximum of four connections on the host machine. The amount of time a connection is held in the cache is customizable, but users should be aware that holding a connection ties up a network port, and users may run out of available Telnet ports on the machine because the RCI has cached them and may not free them for the duration specified in the cache specification file. In such cases, you can still free the ports by making a call to the programmatic interface Release_Unused_Connections. This call releases unused cached connections.

Specifying Remote Login Information

To perform any operation on the remote machine from the host, such as creating libraries, downloading files, or compiling or linking remotely, the RCI must know:

- Which remote machine name to select from the transport name map
- The remote user and password under whose authorization the operation should take place
- The remote directory in which the operation is to take place

Remote Machine Name

The name of the remote machine with which RCI commands in a given view should operate is determined by the first nonnull value in the following lists:

- When you create a view for the first time:
  - Rci.*Custom_Key*_Default_Machine switch (one for each registered target key in a host library) in the host view's enclosing library compiler switches. The RCI traverses the enclosing libraries until it finds a nonnull switch value.
  - Session_Rci.*Custom_Key*_Default_Machine (one for each registered target per session). The RCI uses this value if it does not find a nonnull value in the library compiler switches.

    —*or*—
  - The Remote_Machine parameter in the command being executed, if such a command parameter exists (with package Rci_Cmvc commands.)

  The RCI uses this value to set the Ftp.Remote_Machine switch for the new view.
- When you use a command with an existing view:
  - The Remote_Machine parameter in the command being executed, if such a command parameter exists (with package Rci or Rci_Cmvc commands.)
  - The Ftp.Remote_Machine library switch in the context of the view or object against which the command is taking place.
  - The Session_Ftp.Remote_Machine switch.

If it cannot find a nonnull value, the host does not attempt to connect to a remote machine.

The value of the Session_Rci.Auto_Create_Remote_Directory switch controls whether or not the RCI actually creates a remote directory on the compilation platform when it creates a new host view as the result of a CMVC command. The Rci_Cmvc commands create a remote directory regardless of the value of the Session_Rci.Auto_Create_Remote_Directory switch.

If the RCI finds a value for the remote machine when creating a new view, it sets the library switch Ftp.Remote_Machine for the view in question, even if it does not create the remote directory. The CMVC view-creation commands (Cmvc.Initial, Cmvc.Make_Path, and Cmvc.Make_Subpath) create the remote machine name from the Rci.*Custom_Key*_Default_Machine or Session_Rci.*Custom_Key*_Default_Machine switches. Their corresponding Rci_Cmvc commands (Rci_Cmvc.Initial, Rci_Cmvc-.Make_Path, Rci_Cmvc.Make_Subpath) have a Remote_Machine parameter. Its value must be nonnull if you want to create a remote library. Otherwise, this value must come from the Session_Rci.*Custom_Key*_Default_Machine switch for the registered target key; the RCI uses that value to set the new view's Ftp.Remote_Machine switch.

You can display the Ftp.Remote_Machine switch's current value with the Rci.Show-_Remote_Information command. You can also display the value that the RCI would use if you run one of the view-building commands. Use the Rci.Display_Default-_Naming command to see that information.

Remote Username and Password

The name of the remote user and that user's password under whose authority activities should take place on the remote machine are determined by the first nonnull username value in the following list:

- Library-switch file: The Ftp.Username and Ftp.Password switches in the context of the view against which the command is taking place
- Session-switch file: The user's Session_Ftp.Username and Session_Ftp.Password session switches

- Remote-passwords file: As described in package Remote_Passwords in the Session and Job Management (SJM) book of the *Rational Environment Reference Manual*

These values apply to operations in RCI views that do not have matching compiler switches set. Each line in the remote-passwords file contains a remote machine name, a username, and a password (which may be encrypted):

```
machine1      username1     password1
machine2      "username2"   "password2"
machine4      username4     <DES:018853AB3F000CC4FEB7D5F91C2772FD>
```

In the first two examples, the username and password are not encrypted. The third example shows an encrypted password. You can use commands from package Remote_Passwords to encrypt passwords.

If the RCI cannot find a nonnull username, the host fails to connect with the remote machine.

When you create an RCI view and want to create the associated remote library at the same time, you must set these values in the session-switch file or the file designated by the Profile.Remote_Passwords session switch. (The library-switch file for the view does not yet exist, because the view does not yet exist.)

Remote Directory

The name of the remote directory, in which RCI commands in a given view should execute or send and receive files, is determined by the first nonnull value in the following lists.

- When you create a view for the first time:
  - The name of the remote directory derived by appending the unique library suffix to:
    - Rci.Custom_Key_Default_Roof switch (one for each registered target key in a host library) in the host view's enclosing library compiler switches. The RCI traverses the enclosing libraries until it finds a nonnull switch value.
    - Session_Rci.Custom_Key_Default_Roof (one for each registered target key per session). The RCI uses this value if it does not find a nonnull value in the library compiler switches.

    —or—
  - The Remote_Directory parameter in the command being executed, if such a command parameter exists (with package Rci_Cmvc commands).

  The RCI uses this value to set the Ftp.Remote_Directory switch for the new view.
- When you use a command with an existing view:
  - The Remote_Directory parameter in the command being executed, if such a command parameter exists (with packages Rci and Rci_Cmvc commands).
  - The Ftp.Remote_Directory library switch in the context of the view or object against which the command is taking place (for existing views).
  - The Session_Ftp.Remote_Directory switch.

If a nonnull value cannot be found, the RCI cannot complete the requested remote action and displays a message.

The value of the Session_Rci.Auto_Create_Remote_Directory switch controls whether the RCI actually creates a remote directory on the compilation platform when it creates a new host view as the result of a CMVC command. The Rci_Cmvc commands create a remote directory regardless of the value of the Session_Rci-.Auto_Create_Remote_Directory switch.

The RCI constructs the remote directory name by appending a suffix to a remote roof. It finds the remote roof by traversing the new view's enclosing library for a non-null value in the Rci.*Custom_Key*_Default_Roof switch. If the roof is not found, the RCI uses the value in the Session_Rci.*Custom_Key*_Default_Roof switch.

On the host, for example, assume you want to create a Vax_Vms_Dec_Xt view named !Projects.Space_Shuttle.User_Interface.Console.Rev1_Working on the host. The first nonnull value for an Rci.*Custom_Key*.Default_Roof switch is associated with !Projects.Space_Shuttle and contains the value [Space]. The RCI takes the unique path value for the specified view, User_Interface.Console.Rev1_Working, and appends it to the roof value in the switch. It stores the complete value, [Space.User_Interface-.Console.Rev1_Working], in the format for the compilation platform's operating system and then stores the value in the specified host view Ftp.Remote_Directory switch when it creates the new view.

If you use the Session_Rci.*Custom_Key*_Default_Roof switch, the RCI appends the full pathname to the value in the switch. If the Session_Rci.Vax_Vms_Dec_Xt_Default_Roof contains the value [Space], the RCI creates the name [Space.Projects-.Space_Shuttle.User_Interface.Console.Rev1_Working] and stores it in the Ftp.Remote_Directory switch of the Rev1_Working view. Because it uses the full pathname with the session-switch value, the RCI can create deeply nested names using this method. (Your customizer can also control levels of directory nesting in your extension.)

The Ftp.Remote_Directory switch is initially set for each RCI view when the view is created. The CMVC view-creation commands (Cmvc.Initial, Cmvc.Make_Path, and Cmvc.Make_Subpath) create the remote directory name from the Rci.*Custom_Key*-_Default_Roof or Session_Rci.*Custom_Key*_Default_Roof switches and unique pathname suffix. The corresponding RCI view-creation commands (Rci_Cmvc.Initial, Rci-_Cmvc.Make_Path, Rci_Cmvc.Make_Subpath) have a Remote_Directory parameter, which must be nonnull if you want to create a remote library to match the view; the RCI uses this value to set the new view's Ftp.Remote_Directory switch.

You can display the Ftp.Remote_Directory switch's current value with the Rci-.Show_Remote_Information command. You can also display the value that the RCI would use if you run one of the view-building commands. Use the Rci.Display-_Default_Naming command to see that information.

A Suggested Strategy

Because the RCI is designed to encourage consistency between the host view and the remote library, each view must be associated with exactly one remote library on one remote machine.

The RCI provides for the following strategy:

- Setting Remote_Machine and Remote_Directory values

  Before your create an RCI view and its associated remote directory, you need to establish values for the naming scheme that CMVC uses to build library-switch

values for remote parameters and specify whether or not you want remote directories built when you first create the new host view.

- Set the Remote_Machine value

  Before creating an RCI view and its associated remote library, set the library switch in an enclosing library to contain the value for the Rci.*Custom_Key*_Default_Machine library compiler switch or the RCI Session_Rci.*Custom_Key*_Default_Machine session switch. Use the session switch if you are using a single compilation platform per target key.

- Set the Default_Roof value

  Before creating an RCI view and its associated remote library, set the library switch in an enclosing library to contain the value for the Rci.*Custom_Key*_Default_Roof library compiler switch or the RCI Session_Rci.*Custom_Key*_Default_Roof session switch. Use the library switch to control the value if you are using more than one project per host per target key. Use the library switches to avoid deeply nested directory names.

- Check the Session_Rci.Auto_Create_Remote_Directory value

  If you want to create new remote libraries when you create new host views, confirm that the value of the Session_Rci.Auto_Create_Remote_Directory switch is set to True.

When you create an RCI view, the Environment automatically creates a library-switch file for that view. The RCI sets the values for the Ftp.Remote_Machine and Ftp.Remote_Directory switches from the values in the default naming scheme. If you specify Remote_Machine or Remote_Directory with parameters on the command that creates the view (from package Rci_Cmvc), the library switches are set automatically to those values, regardless of the value of the Session_Rci.Auto_Create_Remote_Directory switch.

- Setting usernames and passwords

  Before creating an RCI view and its associated remote library, you should establish either the session switches for username and password or the Remote_Passwords file (designated by the Profile.Remote_Passwords session switch) to allow access to the remote machine during view and library creation.

  - If all users of a view share the same remote username and password, set Ftp.Username and Ftp.Password to those shared values in the view's switch file with the Switches.Edit or Switches.Set commands.

    —or—

  - If all users log in with their own username on the remote machine, set the values of your username and password in the Session_Ftp.Username and Session_Ftp.Password switches with the Switches.Edit_Session_Attributes.

    —or—

  - If you need password encryption or global control over remote users and passwords, you can use the Remote_Passwords files as described in package Remote_Passwords in the Session and Job Management (SJM) book of the *Rational Environment Reference Manual.*

- Controlling development on more than one compilation platform

  If you are developing projects with more than one compilation platform per target key and must map multiple views of the same subsystem onto those different platforms, you can either:

- Set the Session_Rci.Auto_Create_Remote_Directory switch to False. After you have created the new view, reset the value to True. Use the Rci.Build_Remote_Library command and specify the values for the different compilation platforms in the Remote_Machine and Remote_Directory parameters (or their default library switches).

  —*or*—

- Before you create a view, set the session or enclosing library Rci.*Custom_Key*_Default_Machine switch to the value of the remote compilation platform where you want to build remote directories. To change this value to the new platform, modify the switch value before you create a view.

SETTING UP MODEL WORLDS

This section provides an overview of libraries and model worlds, which are described in more detail in the Project Management (PM) book of the *Rational Environment Reference Manual.*

Features of a Library

Any library structure in the Rational Environment consists of:

- The *library* itself (a set of objects such as Ada units and files). For RCI development, this is a *view.*
- A *target key* that identifies the target and operating system for which code in the library is being developed. It may refer to the Rational Environment on the R1000 itself.
- A *library-switch file* that defines how programs and tools operate in the library.
- A set of external *links* that give the library access to Ada units residing outside the library.

Features of a Model World

When a library is created, you can set up each of its features manually or you can copy a *model world* with previously determined values.

A model world is a library (world) whose directory structure, target key, library-switch settings, and links are copied into the new library structure.

Target-dependent characteristics of the integrated development-cycle environment (such as the language characteristics of package Standard) are set by choice of a model world for that target. The native R1000 environment also is specified by a model world. Predefined models are provided as described in "Predefined Model Worlds," below. You can create new project-specific models using a copy of an existing model.

Target Keys

The target key determines target-specific aspects of compilation.

Target keys for the RCI are:

- R1000 (supplied by Rational)
- *Custom_Key* (defined by customizing to create an extension)—for example, I186_Vms_Ddc
- Reserved custom keys (for Rational precustomized RCI extensions):
  - I386_Unix_Als_Xt
  - I486_Unix_Als_Xt
  - I960_Vms_Tar_Xt
  - Rs6000_Aix_Ibm
  - Sparc_Sun_Xt
  - Vax_Vms_Dec_Xt

The RCI target key is composed of three components separated by underscores:

- Target architecture: The architecture of the final target—for example, I186
- Compilation-platform operating system: The operating system that controls the Ada compilation system on the remote compilation platform—for example, Vms
- Compiler vendor: The abbreviation of the target Ada compiler vendor on the remote compilation platform—for example, Ddc

This forms a complete target key—for example, I186_Vms_Ddc.

Only Rational-supported extensions contain the fourth component, Xt.

Table 2-1 lists recommended abbreviations for these compiler vendors.

Table 2-1 Abbreviations for Compiler Vendors

Target Compiler Vendor	**Abbreviation**
Alsys	Als
DDC-I, Inc.	Ddc
Digital Equipment Corporation	Dec
International Business Machines Corporation	Ibm
Sun Microsystems Incorporated	Sun
Tartan Laboratories Incorporated	Tar
Telesoft	Ts
Verdix Corporation	Vdx

The maximum length of a target key is 16 characters, including underscores.

The structure of the target key distinguishes between compilers on the same compilation platform; for example, between an AIX™-based IBM compiler and an AIX-based Verdix® compiler. This convention ensures uniqueness among possible RCI target keys.

The form described above defines the compilation portion of the target-key name. In addition to the compilation portion, some target keys (called *design targets*) contain a design portion. The design portion specifies a program design language (PDL)

to be used. For further information on PDL and design targets, refer to the *Rational Design Facility: DOD-STD-2167* (or 2167A) *User's Manual.*

The RCI now supports RDF/RCI composite target keys, which means that you can use both the RDF and the RCI in the same view.

RDF views should be built first using Design.Initial, without composite keys, to establish the desired design hierarchy. Once you have the hierarchy, you should create new views using the composite target key.

Use these steps to create views using a composite target key:

1. Run What.Users to verify that the RDF and the RCI are elaborated. Check for the PDL Registration job, the Rci_Compiler job, and the RCI Extensions job.
2. If a composite model world does not already exist, use Design.Create_Model to create one. For example:

```
Design.Create_Model
       (Compiler_Model        => "Rs6000_Aix_Ibm",
        Design_Facility_Model => "Rational_2167a_R1000",
        New_Model_Name        => "A_New_Model");
```

   creates a model world called A_New_Model with the following elements:
   - The RDF directory structure
   - The RCI links
   - A composite target key, in this case Rational_2167a_Rs6000_Aix_Ibm
3. Create a new view, using Cmvc.Make_Path (or Rci_Cmvc.Make_Path), specifying the composite model name in the Model parameter.

Once you create a composite view, both RDF and RCI semantics are performed when you promote any unit in that view to the installed state.

Note that system views cannot be combined views. This means that you cannot move any RDF hierarchy component that resides in a system view to a composite view. This is not a major problem for RCI users since no actual code implementation takes place in system views. RDF document generation is supported since it does not require the compiler portion of the target key to be consistent across all views. That is, views with different target keys can be mixed in your activity for document-generation purposes as long as the RDF portion of the keys is consistent.

Library-Switch Files

A default library-switch file may be associated with each model world; the switch settings for a particular directory or world are editable. Switch names, values, and effects related to RCI use are introduced in "Setting Session and Library Switches" in Chapter 3 and discussed in the Library Management (LM) book of the *Rational Environment Reference Manual.*

Links

A set of default links is associated with each model world. These mappings of Ada units to simple names are editable objects that are described in the Library Management (LM) book of the *Rational Environment Reference Manual.*

The links in a model can be used to control the portability of code; if links exist only to portable objects, then *with* clauses cannot reference specific objects in the Environment that do not exist on the compilation platform.

Predefined Model Worlds

The models supplied by Rational for use in the native R1000 reside in the world !Model. They are:

- R1000: Includes links for R1000-specific facilities and sets the target key to R1000
- R1000_Portable: Includes links for only those facilities specified by the Ada LRM to ensure portability, such as Text_Io and Calendar, and sets the target key to R1000

The model for use in the integrated development cycle also normally resides in the !Model world. The actual target-key name is determined by the customization extension; this manual uses the name:

- *Custom_Key:* Includes links for target-dependent facilities as well as Ada LRM-predefined units and sets the target key to *Custom_Key*

Creating a Project-Specific Model World

Users can optionally create their own model worlds to suit their particular needs. When R1000 library structures are created, either a Rational-supplied or project-specific model world can be used. A project-specific model can be created by modifying a copy of a Rational-supplied or *Custom_Key* model, as follows:

1. Create a new *Custom_Key* model world with Library.Create_World, using !Model-.*Custom_Key* for the Model parameter. The new model should be located somewhere other than in the !Model directory to avoid potential naming conflicts with future releases of Rational target tools. A directory specific to the project for which the new world will be used is a preferred location.
2. Use the Switches.Edit command to customize the model world's switch file. Edit the switch file to set library switches with values specific to the project.
3. Use the Links.Edit command to customize the model world's links. For RCI views, the model links must refer only to units in the predefined world. The model can reference only units that are accessible to all libraries on the target machine. The predefined world is the only place on the R1000 to locate those universally accessible units.
4. Customize the model world's directory structure as necessary.

You can use this new *Custom_Key* model world as the Model parameter when creating RCI views from R1000 views. You must specify the full pathname to the *Custom_Key* model world. The new model can be used for any number of RCI views.

PREPARING TO SET UP LIBRARY STRUCTURES

The RCI environment for native R1000 and integrated development cycles consists of library structures (subsystems, paths, views, and directories) and of related library structures on the remote compilation platform. These structures contain the software being developed.

Ada units for the chosen target must be compiled within a view with the *Custom_Key* target key.

Note: *If you have not already read "Setting Up the Rational Environment" on page 6 in Chapter 1, do so before continuing with this section.*

If you are not familiar with Rational's system of configuration management and version control (CMVC), you may want to have available a copy of the Project Management (PM) book of the *Rational Environment Reference Manual.*

Overview

To set up the appropriate library structure, you must understand the concepts behind subsystems and views. These are discussed below in "Understanding Subsystems" and "Comparing Types of Views."

You must then decide under what software environment you want to develop code. (See "Choosing Library Structures," below.)

Once you have decided, you can create appropriate subsystem views, which is discussed in "Setting Up Library Structures" on page 35.

Understanding Subsystems

The library structures required for development under the RCI are subsystems, in which native R1000 code and target code are developed in working views of different paths. Subsystems in conjunction with CMVC and the RCI extensions to CMVC operations (called cross-system consistency management) automate much of the development process. This includes coordinating host and remote versions of the software, maintaining development history, and synchronizing development among many developers.

For any *working view* in a subsystem, there is a *path* that includes that working view and the *releases* generated from it (see Figure 2-1). Development that takes place in the working view can be referred to as taking place in the path.

In addition, the RCI allows maintenance of a library structure on the compilation platform whose contents parallel the contents of the RCI subsystem view on the R1000. For example, Figure 2-2 (page 30) shows a remote library structure that might be used to map to the host library structure shown in Figure 2-1. See the "Managing Remote Libraries" section on page 89 in Chapter 6 for further information.

For details about subsystems, refer to the Project Management (PM) book of the *Rational Environment Reference Manual.*

Development in subsystems can use the facilities provided by Rational's system to manage project development. For integrated development, a subsystem will probably contain a native R1000 path and at least one path per target. Typically, host development is done in a working view with an R1000 target key; changes then are accepted into a working view with a *Custom_Key* target key.

The CMVC system described in the PM book provides sufficient functionality to perform the cross-system library and unit management required for the RCI. Package Rci_Cmvc provides several corresponding commands that you can use in place of similarly named commands in package Cmvc. The additional commands supplement the CMVC functionality if you are unable to use the default remote directory and machine-naming system by providing command parameters to specify those values directly.

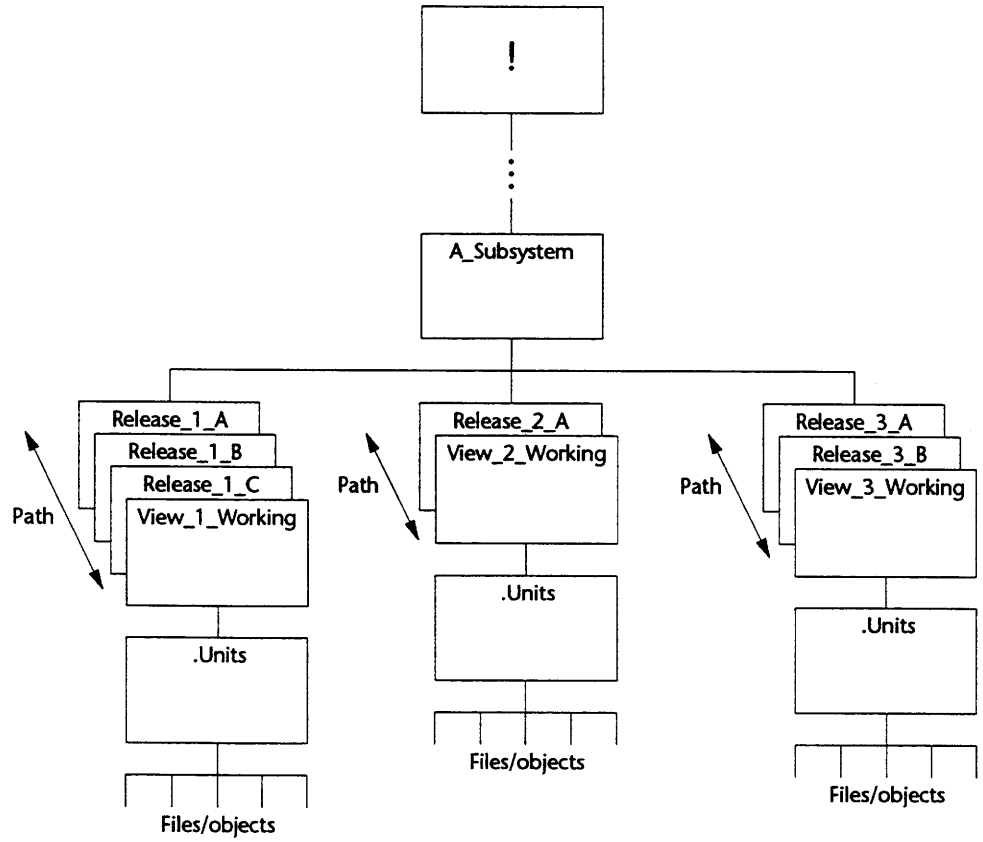

Figure 2-1 Library Structure Showing Subsystems, Views, Paths, and Releases

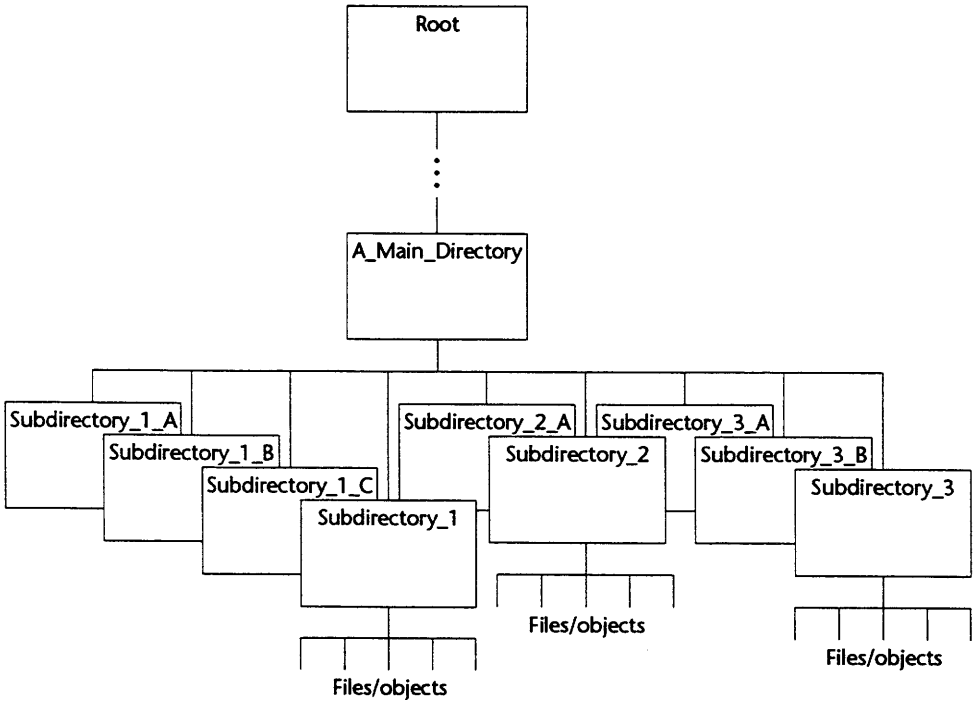

Figure 2-2 Remote Library Structure

Comparing Types of Views

The Rational Environment and the RCI use different types of views during the integrated development cycle.

Three types of views are available in Rational Environment subsystems:

- Code view
- Spec/load view
- Combined view

For details about types of views and the choice between them, see the Project Management (PM) book of the *Rational Environment Reference Manual.*

When a view of any type is discussed, it is referenced by the environment in which code is developed; therefore an *R1000 view* has an R1000 target key and is completely under control of the Rational Environment; an *RCI view* has a *Custom_Key* target key and is under RCI control.

R1000 Views

Because development in R1000 views is intended as the initial step in creating code to execute on a remote compilation platform, it is expected that you will copy code in these views into RCI views and the code should therefore be joined between the views. Therefore, it is strongly recommended that you do development in R1000 paths in subsystems and views rather than worlds and directories.

When possible, it is best to use spec-view/load-view pairs for their advantages in minimized recompilation requirements and flexible recombinant testing.

RCI Views

In RCI paths, combined views are the only valid type of view; spec/load views are not allowed. Worlds are not supported. This occurs because the RCI's goal is to map host views onto remote libraries, and it is difficult to model the Rational dynamic importing model used by spec/load views onto most remote library structures. Most target compilation systems require users to compile directly against the code with which they will link.

For this reason, you may want to develop code in spec/load views for the native R1000 development cycle and move it into combined views for the integrated development cycle.

The RCI allows limited use of spec views. See "Creating an RCI Spec View" on page 40 for more information.

Choosing Library Structures

Before creating library structures on either the host or the remote compilation platform, you should have some idea of:

- Whether you will use the R1000 native development cycle
- Whether you have target-independent code whose target-independence should be enforced by Rational Environment facilities

- Whether you want to maintain an exact duplicate of all library structures, including releases, on the compilation platform or only the minimum required libraries
- The scope of the project and how it divides into logical subcomponents

Using the R1000 Native Development Cycle

This development cycle has all of the advantages described in the "Developing and Executing Code for the Target Machine" on page 11 in Chapter 1.

This is, however, a recommended and not a required portion of the development cycle for target code.

Views with R1000 target keys are created and maintained using standard CMVC commands.

Enforcing Target Independence

When creating R1000 views, you have the option of using either the !Model.R1000 model world, which includes links to R1000-specific subprograms, or the !Model-.R1000_Portable model world, which provides links only to standard LRM-predefined routines.

There is no *Custom_Key*_Portable model world; the effect is maintained by joining units in RCI views that should remain portable with units in the R1000_Portable view. Then you can update and test these units in the R1000 view before you accept changes into their joined files in the RCI view.

Duplicating Structures on the Compilation Platform

Remote library structures are created automatically by any package Cmvc or Rci-_Cmvc view-creation commands, such as Initial and Release, if the Session_Rci-.Auto_Create_Remote_Directory switch is True and remote library management has been *registered* with the RCI. Registering remote library management is described in "Enabling Remote Extensions Management" on page 18. Discussions in this chapter assume that registration has taken place.

Remote directories and program libraries that parallel working views must exist for remote compilation to take place. Several rules apply to remote-to-host library mapping; these rules are given in "Limitations and Restrictions" on page 85 in Chapter 6.

Creating a release library on the compilation platform when a view is released on the host is optional; since the Rational host is being used to take advantage of its CMVC features, you may decide to maintain releases in only one location. Or you may decide to have a copy of only the most recent release for each path on the compilation platform. See "Creating Releases of Views" on page 96 in Chapter 6 for details.

Dividing a Project into Logical Subcomponents

Generally, each major section of a project is placed into its own subsystem. Each subsystem can contain many views, including one view each for R1000-specific, target-independent, and target-dependent code, or an additional set of these views for each subcomponent in the subsystem.

Examples of Library Structures

Here are two possible scenarios for library structures. They by no means define all possible or desirable library methodologies, but they offer suggestions to consider when you design your project structures.

A Simple Example

A minimum library configuration for a simple RCI project might look as shown in Figure 2-3. This project consists of an Ada main unit, Unit_A, that is target-independent. It calls a subprogram that makes use of target-dependent features, such as system calls or a non-Ada language. This unit is written once using Rational Environment subprograms for testing on the host (Unit_B) and is rewritten for the integrated development cycle using the target's features (Unit_C). Since there is only one target view, only one directory needs to exist on the compilation platform to contain copies of Unit_A and Unit_C.

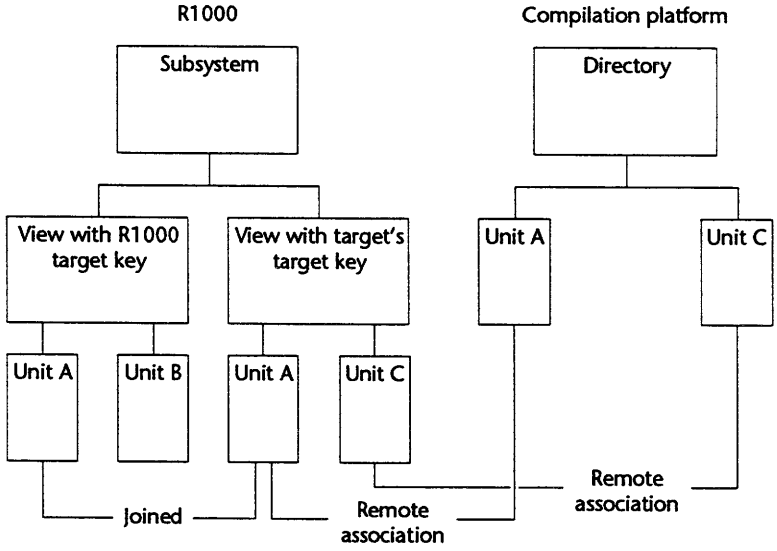

Figure 2-3 Simple Host and Remote Library Structures

A More Complex Example

In the example shown in Figure 2-4, portable and target-dependent code are treated as separate development projects and are therefore placed in separate subsystems.

Within each subsystem, there is a view to use in the R1000 native development cycle and one to use in the integrated development cycle. Note that units in joined portable views should have a one-to-one relationship, but units in joined target-dependent views may be the same or vastly different. For example:

- Unit_A might be essentially the same for execution on the host and on the compilation platform but might call different Standard packages.
- Unit_B might be written using Rational Environment system calls, and Unit_C, although it performs the same function, might be written using assembly language for the target machine.

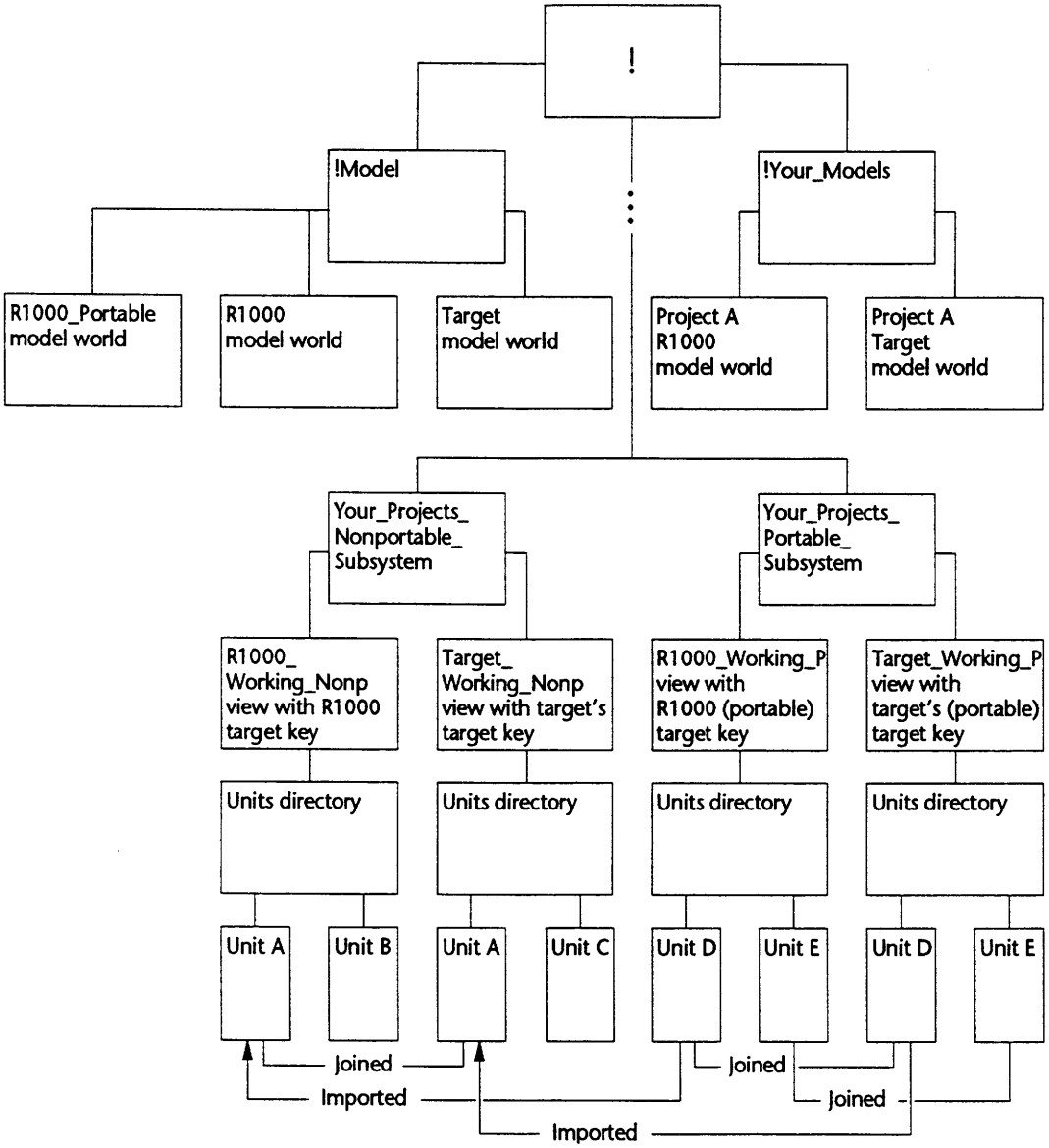

Figure 2-4 Subsystems for Integrated Compilation

- Unit_D and Unit_E might be written using target-independent LRM standard code, so the units are identical in Rev1_Working and Target_Working.

If the Target_Working view under Your_Projects_Portable_Subsystem was the first view to be created along with the subsystem, the command to create them and the associated remote library, as shown in Figure 2-5, might be:

```
Cmvc.Initial
     (Subsystem              => "Your_Projects_Portable_Subsystem",
      Working_View_Base_Name => "Target",
      Model                  => "!Model.Custom_Key");
```

where the default naming scheme or the Remote_Directory parameter of Rci-_Cmvc.Initial provides the remote directory /root/.../portable.

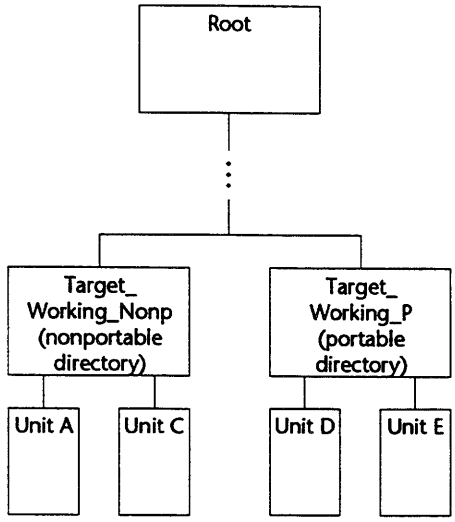

Figure 2-5 Remote Libraries Corresponding to Subsystem Views

SETTING UP LIBRARY STRUCTURES

Before proceeding with this section, be sure to read the preceding section, "Preparing To Set Up Library Structures."

There are three structures that can be set up separately, although the RCI automatically creates most of the necessary structures:

- Subsystems
- Views: The first view in each subsystem is created automatically when the subsystem is created.
- Remote libraries: These are created automatically any time that a view with a *Custom_Key* target key is created if remote library management is registered with the RCI as described in "Enabling Remote Extensions Management" on page 18.

Creating a Subsystem and an R1000 View

If you plan on initiating development in a host development cycle, use commands from package Cmvc to create a subsystem whose first view has an R1000 target key. To skip the native development cycle, or to create a subsystem that initially contains only target code, use package Cmvc commands with the necessary RCI target key.

To prepare native R1000 development views in a subsystem:

1. To create the desired subsystem, enter the Cmvc.Initial command in a command window and press [Complete].
2. Specify the desired model world, either R1000 or R1000_Portable, using the Model parameter of the Initial command so that the initial working view in each subsystem has an R1000 target key.

3. Specify the base-name prefix for the first view in the subsystem, according to the project's naming conventions. The view name is constructed by appending `_Working` to the base name. For example, the pathname can indicate the path's target and whether it is a development or production view, as in:

   `Working_View_Base_Name = "Rev3_`*Custom_Key*`_Devel"`

   This creates a view called Rev3_*Custom_Key*_Devel_Working.
4. Press [Promote]. The subsystem and view are created. This initial working view provides a native R1000 development path.
5. Go to the Units library in the working view of the subsystem that you have just created.
6. Create Ada units in the Units library, and develop and test them as much as possible before creating the target paths.

Displaying Defaults for Remote RCI Names

Before you use CMVC to create a new RCI view and remote directory structure, you can display the remote machine and directory name that CMVC will choose for a potential view with the given target key. These values are based on the current values defined by the default switch-naming scheme described in "Specifying Remote Login Information" on page 20.

To display the values:

1. Enter the Rci.Display_Default_Naming command and press [Complete]:

```
Display_Default_Naming
        (Potential_View : String = "",
         Target_Key     : String = "",
         Response       : String = "<PROFILE>");
```

2. Enter values for the following parameters:
   - Potential_View: Specifies the name of the view you want to create using the CMVC view-creation command.
   - Target_Key: Specifies the target key for the view you want to create.
3. Press [Promote]. The command displays the values of Remote_Machine and Remote_Directory that CMVC would use to create the new view.

Creating a Subsystem and an RCI View

If a subsystem already exists and you need only to add a view, see "Creating Additional R1000 Views" and "Creating New RCI Views," below. Usually, R1000 views and RCI views will exist in the same subsystem.

To create a new subsystem with an RCI view in it:

1. If you want to create the remote directory that is associated with this view at the same time that you are creating the view:
   - Verify that your Session_Ftp.Username and Session_Ftp.Password session switches are set to provide appropriate access to the remote machine for

library creation, or that the equivalent information is supplied in the Remote_Passwords file as described in "Remote Username and Password" on page 21.

- Verify that the Session_Rci.Auto_Create_Remote_Directory switch is set to True.
- If you are using CMVC commands, verify that the RCI default roof and remote machine switches are correctly set as described in "Specifying Remote Login Information" on page 20.
- Verify that remote library management has been enabled as described in "Enabling Remote Extensions Management" on page 18.
- Verify the network-setup information as described in "Setting Up Remote Communications" on page 19.

2. Enter the Cmvc.Initial command in a command window and press [Complete].
3. Enter the name of the new subsystem into the Subsystem parameter.
4. Specify at least the following parameters:
   - Working_View_Base_Name: Specify the base name for the first view in the Working_View_Base_Name parameter; the view name is constructed by appending `_Working` to the base name.
   - If you are using Cmvc.Initial, the default naming scheme controlled by the default switches provides the Remote_Directory and Remote_Machine values. If you are using Rci_Cmvc.Initial, complete the following parameters:
     - Remote_Machine: Specify the name of the remote machine on which a directory and program library matching the view should be constructed as described in "Specifying Remote Login Information" on page 20. For example:

       ```
       Remote_Machine = "Tilden"
       ```

     - Remote_Directory: This is used to construct the appropriate remote library as described in Chapter 6, "Library Management." It may be appropriate to give the remote library the same name as the host view. For example:

       ```
       Remote_Directory =>
       "/usr/project1/rev3_Custom_Key_devel_working"
       ```

     The *remote library* consists of the named directory, which contains a program library and a set of imports; it will also eventually contain Ada units. If Remote_Directory or Remote_Machine is undefined in switches (for Cmvc commands) or left blank (for Rci_Cmvc commands), the RCI does not create the remote library, and Ada units in the view cannot be promoted past the installed state.
   - Subsystem_Type: This can be Cmvc.Spec_Load_Subsystem (for R1000 views) or Cmvc.Combined_Subsystem.
   - Model: Specify the full pathname of a model that has a *Custom_Key* target key. For example:

     ```
     Model => "!Model.Custom_Key"
     ```

5. Press [Promote]. The remote library is created along with the view, as described in the "Management Of Remote Libraries" section of Chapter 6, "Library Management."

This working view provides an RCI development path.

Creating Additional R1000 Views

To create a new R1000 view within a subsystem, use the Cmvc.Make_Path command, as described in the Project Management (PM) book of the *Rational Environment Reference Manual.*

Creating New RCI Views

RCI views are created either by copying an existing view or by creating a new, empty view.

To create an empty RCI view in an existing subsystem, use the Cmvc.Initial (or Rci_Cmvc.Initial) command as described above, selecting the existing subsystem rather than naming a new subsystem.

To create an RCI view from an existing R1000 view or another RCI view:

1. If you want to create the associated remote library at the same time as the view is created, see step 1 in "Creating a Subsystem and an RCI View," above.
2. Use the Cmvc.Make_Controlled command to place the units under CMVC control; only these units will be joined between the new RCI path and the original R1000 path.
3. Make sure all of the units in the R1000 or RCI view are checked in using the Cmvc.Check_In command.
4. Use the Cmvc.Make_Path (or Rci_Cmvc.Make_Path) command to create a target path and a unique matching directory on the remote machine, as described in the "Choosing Library Structures" on page 31. To do this, select the working view of the R1000 or RCI path. Enter the Cmvc.Make_Path command and press **[Complete]**. The command looks like this:

```
Cmvc.Make_Path
    (From_Path             : String  := "<CURSOR>"
     New_Path_Name         : String  := ">>PATH NAME<";
     View_To_Modify        : String  := "";
     View_To_Import        : String  := "<INHERIT_IMPORTS>";
     Only_Change_Imports   : Boolean := True;
     Create_Load_View      : Boolean := False;
     Create_Compined_View  : Boolean := False;
     Model                 : String  := "<INHERIT_MODEL>";
     Join_Paths            : Boolean := True;
     Remake_Demoted_Units  : Boolean := True;
     Goal Compilation.Unit_State
                                     := Compilation.Coded;
     Comments              : String  := "";
     Work_Order            : String  := "<DEFAULT>";
     Volume                : Natural := 0;
     Response              : String  := "<PROFILE>");
```

5. Specify the New_Path_Name parameter as described for the Working_View_Base_Name parameter, above.
6. For Rci_Cmvc.Make_Path, fill in the Remote_Machine and Remote_Directory parameters as described above.
7. Fill in the Model parameter as described above.

8. Set Join_Paths to True if all or most of the controlled units are to be shared (joined) between paths on the host; specify False if none or few of the units are to be joined. (When corresponding units are joined, changes to one unit can be propagated automatically to the others in its join set. Furthermore, a joined unit can be checked out in only one path at a time. Units that are not joined can be checked out and edited concurrently.)
9. Change imports to reference RCI views instead of R1000 views. Figure 2-6 helps to illustrate when this is necessary.

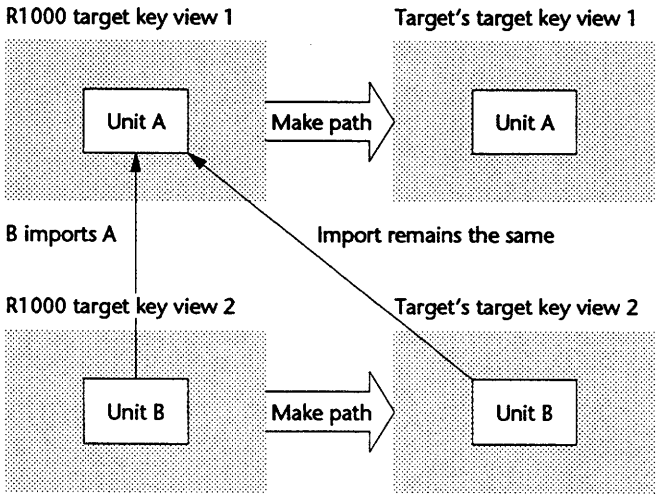

Figure 2-6 Make_Path Uses Same Imports

Because the Make_Path command creates a new view, it copies, *not changes*, the names of other views from which that view imports subprograms. Hence, your new target view will still try to import subprograms from a view with an R1000 target key rather than from your new view.

Note: *Because RCI views can import only other RCI views with the same target key, this causes an error.*

Such imports must be changed by you, either by specifying the View_To_Import in the Cmvc.Make_Path (or Rci_Cmvc.Make_Path) command or afterward with the Cmvc.Import command. If you are developing in more than one subsystem (very likely), you can specify only one naming expression for imports with the Make_Path command and will have to change others afterward (see step 11).

10. Press [Promote]. All units are copied from the R1000 or RCI path to the RCI path, and controlled units are joined between paths.

    If a connection can be made to the remote machine, the RCI creates an appropriate remote library as described in Chapter 6, "Library Management." If the remote library cannot be created or located, the RCI issues a warning and the host units cannot be promoted to the coded state. You will have to create the remote library explicitly as described in Chapter 6 before units can be promoted to the coded state.

    Host units are promoted to the indicated goal state if possible. Promoting the units to the coded state causes the RCI to download files to the remote library and compile them as described in "What Happens During the Coding Step" on page 59 in Chapter 3. The remote files are named as described in "Remote Files and Names" on page 56 in Chapter 3.

11. Use the Cmvc.Sever or Cmvc.Join commands as necessary, so that all target-independent units are joined between the two paths and target-specific units are severed.
12. Add remaining imports as described in step 8.
13. Use Switches.Edit to change settings in the library-switch file if needed. See "Setting Session and Library Switches" on page 45 in Chapter 3.

The resulting RCI path consists of a working combined view that contains a copy of the units from the R1000 or RCI path. Development can continue in either path, as appropriate.

If additional units are developed that need to be copied between views, see "Consistency Between Views On The Host" on page 75 in Chapter 5, and then use the Cmvc.Sever and Join commands as needed.

If additional units are created or edited on the remote machine, see Chapter 5, "Maintaining File Consistency," for information on maintaining consistency with the host units.

Creating an RCI Spec View

In most cases, you use combined views only, since most target compilation systems require that you compile directly against the code with which they link, invalidating the spec-load approach. The R1000 compilation system is unique in the sense that you can compile against spec views and not specify corresponding load views until link time, using activities.

The RCI provides limited use of spec views. Spec views represent placeholders for remote libraries. Creating a spec view does not create a new target directory, and compiling in a spec view does not cause remote compilation to take place. Spec views should be spawned from preexisting combined views that have remote libraries associated with them. A spec view spawned from a combined view (in a spec-load subsystem) automatically obtains the same target key and remote information (machine and directory) as the combined view. You can compile and link against a spec view, even if the combined view it was derived from is no longer present, as long as the remote library still exists.

Note that because of the placeholder nature of spec views, it is possible to create them on one R1000, use Archive.Copy to copy them to new R1000s, and use them as you would on the original. When working in RCI views, it is not necessary to have Ada bodies in the entire execution closure located on each R1000, but you must represent each remote library by at least a spec view. It is important to note, however, that the RCI requires each remote library in the execution closure (not including those represented in the predefined world) to be represented by a view on each new R1000.

In general, to spawn spec views for a set of subsystems, use the Cmvc.Make_Spec_View (or Rci_Cmvc.Make_Spec_View) command as follows:

1. Enter the Cmvc.Make_Spec_View command and press [Complete] to modify the following parameters:

```
Cmvc.Make_Spec_View
     (From_Path           => "<CURSOR>",
      Spec_View_Prefix    => ">>PREFIX<<",
      View_To_Import      => "<INHERIT IMPORTS>",
      Only_Change_Imports => True;
```

2. Enter the value for the following parameters:
   - From_Path: The set of views—for example, `[S1.<combined_view1>, S2.<combined_view2>, ... SN.<combined_view3>]`
   - Prefix: The prefix of the spec view—for example, `Rev1`
   - View_To_Import: `[S1, S2, S3 ... SN]` (all subsystems in closure)
3. Press [Promote]. The command creates the new spec view.

For example, if the following view structure exists on R1000A:

```
Subsystem       View                  Imports
-------------------------------------------------------
S1              S1.Rev1_Working       S2.Rev1_Working
S2              S2.Rev1_Working       S4.Rev1_Working
S3              S3.Rev1_Working       S4.Rev1_Working
S4              S4.Rev1_Working
```

and you use Cmvc.Make_Spec_View (or Rci_Cmvc.Make_Spec_View) to spawn a spec view, the new subsystem structure after spec views had been generated would look like this:

```
Subsystem       View                  Imports
-------------------------------------------------------
S1              S1.Rev1_Working       S2.Rev1_Working
                S1.Rev1_Spec          S2.Rev1_Spec
S2              S2.Rev1_Working       S4.Rev1_Working
                S2.Rev1_Spec          S4.Rev1_Spec
S3              S3.Rev1_Working       S4.Rev1_Working
                S3.Rev1_Spec          S4.Rev1_Spec
S4              S4.Rev1_Working
                S4.Rev1_Spec
```

If you simply make a spec view S1.Rev1_Spec and archive it to R1000B, you must take additional steps in order to compile and link against S1.Rev1_Spec on R1000B. This is because views compiled against S1.Rev1_Spec may need remote information about the execution closure for target operations. When spawning a spec view from a combined view, it is best to spawn an additional spec view for every subsystem in the closure of the source combined view. The new spec views should import each other in a manner that parallels the source combined view imports. Note that for any new importing spec view, it is not necessary for units in the imported spec view to be equivalent on machine B. Every subsystem is represented, but only by a spec view. The spec views you have copied to machine B are as follows:

```
Subsystem       View                  Imports
-------------------------------------------------------
S1              S1.Rev1_Spec          S2.Rev1_Spec
S2              S2.Rev1_Spec          S4.Rev1_Spec
S3              S3.Rev1_Spec          S4.Rev1_Spec
S4              S4.Rev1_Spec
```

Now it would be possible to compile against any of these spec views just as if they were their source combined views.

Having the spec views import each other in the appropriate manner ensures that the RCI knows all remote libraries in the execution closure of the top-level spec view. If you build a new combined view that imports the top-level spec view, the RCI can do a remote import of the entire remote execution closure of the new view, which is necessary for some target compilers.

Note that CMVC only permits you to spawn spec views in spec-load subsystems. Those users who have done their previous RCI work in combined views need to convert to spec-load subsystems to use spec views. Note also that when using spec-load subsystems, you may need to set Create_Combined_View to True for the various view-creation operations, because the default behavior in spec-load subsystems is to create load views.

3 Getting Started

This chapter assists with the verification of setup steps, explains the differences between native R1000 and remote compilation, and steps through the integrated compilation process from creating code to executing and debugging it on the compilation platform.

Developing source code in subsystems for a target using the RCI is basically the same as developing native R1000 source code using the Rational Environment and CMVC.

There are a number of important differences in developing Ada units in the host and target environments, depending on the nature of the project. These differences are discussed in "Comparing Features Of Rational Environment And Rci Compilation" in Chapter 1.

The steps the RCI takes during the compilation process vary between interactive and batch mode. This chapter describes the steps that you use in interactive mode. Chapter 4 describes batch compilation.

Specifically, this chapter describes the following:

- Setting up your integrated compilation environment
  - Verifying the R1000 library setup
  - Verifying the remote library setup
  - Setting session and library switches
- Creating an Ada program for remote compilation
  - Creating an Ada main unit
  - Using pragma Main
  - Using representation clauses
  - Using implementation-dependent pragmas
- Remotely compiling and linking a simple Ada program
  - Creating an executable program
  - Output from the RCI compiler and linker
  - What happens during the installing, coding, and linking steps
- Demoting a unit

SETTING UP YOUR INTEGRATED COMPILATION ENVIRONMENT

Although the system administrator may have set up your library structure properly for remote compilation, you should verify that everything is in order. In addition, there are library-specific files and switches that you can alter to suit your needs.

Setup steps are the following, discussed in separate subsections below:

- Verifying the R1000 library setup
- Verifying the remote library setup
- Setting session and library switches

Verifying the R1000 Library Setup

No matter what your final target will be, you often compile initially for the native R1000 target. You create code and test it in a library structure with an R1000 target key. The R1000 target key is not shown on your screen (see Figure 3-1).

```
!Users.Rh.Planetary_Motion_Rci : Subsystem Vol 4 { 0};
  Configurations              :   Directory 92/09/04 13:32:41 Rh      Vol 4 { 0}
  Rev1_R1000_Working          :   Load_View 92/09/04 10:54:08 Rh      Vol 3 { 0}
  Rev1_Rs6000_Aix_Ibm_Working :   Comb_View 92/09/04 13:32:33 Rh      Vol 3 { 0}
  State                       :   Directory 92/09/04 10:53:59 Rh      Vol 4 { 0}

= !USERS.RH.PLANETARY_MOTION_RCI (library)        World

!Users.Rh.Planetary_Motion_Rci.Rev1_R1000_Working : Load_View Vol 3 { 0};
  Exports :   Directory 92/09/04 10:54:08 Rh      Vol 3 { 0}    ;
  Imports :   Directory 92/09/04 10:54:07 Rh      Vol 3 { 0}    ;
  State   :   Directory 92/09/04 10:54:23 Rh      Vol 3 { 0}    ;
  Units   :   Directory 92/09/04 11:07:19 Rh      Vol 3 { 0}    ;

= ...MOTION_RCI.REV1_R1000_WORKING (library)        World
```

Figure 3-1 Window with R1000 (Invisible) Target Key

Once the native R1000 development cycle is complete, you will copy your code into a library with a target key appropriate to your desired target (see "Setting Up Library Structures" on page 35 in Chapter 2). Remote compilation and remote linking take place from this new library structure. A window with a *Custom_Key* target key in the banner at the base of the window is shown in Figure 3-2.

Note: *All sections in this and later chapters assume that you have already copied your Ada units into, and are working in, an RCI view.*

```
!Users.Rh.Planetary_Motion_Rci : Subsystem Vol 4 { 0};
  Configurations              :   Directory 92/09/04 13:32:41 Rh      Vol 4 { 0}
  Rev1_R1000_Working          :   Load_View 92/09/04 10:54:08 Rh      Vol 3 { 0}
  Rev1_Rs6000_Aix_Ibm_Working :   Comb_View 92/09/04 13:32:33 Rh      Vol 3 { 0}
  State                       :   Directory 92/09/04 10:53:59 Rh      Vol 4 { 0}

= !USERS.RH.PLANETARY_MOTION_RCI (library)        World

!Users.Rh.Planetary_Motion_Rci.Rev1_Rs6000_Aix_Ibm_Working : Comb_View Vol 3 { 0
  Exports :   Directory 92/09/04 13:32:33 Rh      Vol 3 { 0}    ;
  Imports :   Directory 92/09/04 13:32:33 Rh      Vol 3 { 0}    ;
  State   :   Directory 92/09/04 13:33:04 Rh      Vol 3 { 0}    ;
  Units   :   Directory 92/09/04 13:32:59 Rh      Vol 3 { 0}    ;

= ...RCI.REV1_RS6000_AIX_IBM_WORKING (library)        RS6000_AIX_IBM World
```

Figure 3-2 Window with Rs6000_Aix_Ibm Target Key

You can also confirm your target key by executing this command in a command window:

```
Compilation.Show_Target_Key
```

The Rational Environment then displays a message similar to the following in the current response-profile location (usually a message window):

```
Target key for !USERS.RCI_DEMO.PLANETARY_MOTION.
Rs6000_Aix_Ibm_COMBINED_WORKING is Rs6000_Aix_Ibm
```

The programmatic equivalent of the above is a function that returns the image of the current target key as a string:

```
Compilation.Get_Target_Key
```

If the message window does not show the appropriate target key, then you may need to create subsystems and/or views as described in "Setting Up Library Structures" on page 35 in Chapter 2.

Verifying the Remote Library Setup

See Chapter 6, "Library Management"

Setting Session and Library Switches

Certain RCI options are controlled by switches that manage the characteristics of your working conditions on the Rational Environment. The Rational Environment maintains session switches for each user in the user's home world. A user can have any number of session-switch files, but only one session-switch file is used per session. See the Session and Job Management (SJM) book of the *Rational Environment Reference Manual* for a detailed discussion of session switches.

Other RCI options are controlled by switches that are part of an extensive system of library switches. Library switches are maintained in each view in the library switch file that is created automatically when the view is created. See the Library Management (LM) book of the *Rational Environment Reference Manual* for a detailed discussion of library switches.

The switches that affect RCI operation can be described in the following sets:

- Session and library switches that control the behavior of the RCI and target compiler, labeled *Session_Rci* and *Rci,* fall into three categories:
  - Switches that affect the behavior of the RCI independent of the target compiler. The customization-independent RCI library switches appear in all RCI library-switch files and apply to RCI features common to all RCI extensions. The customization-independent RCI session switches control the creation of remote directories. These switches are shown in Table 3-1.
  - Switches that affect the construction of remote directories for a specific customization. These include session and library switches containing the customization-dependent string (for example: I186_Vms_Ddc_ or *Custom_*). These switches are shown in Table 3-2.
  - Switches whose names begin with a customization-dependent string to specify options for the target compiler. Some of these cause the target compiler to behave in a manner that affects the output of the RCI. Samples of these switches are shown in Table 3-3. A blank table is provided in Appendix C for the customizer to fill in the actual switches defined in your extension.
- Switches used to enable RCI communication with the remote machine, labeled *Ftp,* are shown in Table 3-4. The RCI does not use the similar switches available in the user's session-switch file.

Switch defaults are usually set to match the target-compiler defaults and to minimize the number of files created. At a minimum, you may want to:

- Set switches to define target-compiler operations.
- Set customization-independent switches to control host operations.
- Set the switches described in "Setting Switches for Remote Communication" on page 50.
- Optionally set the switches described in the "Saving Assembly Source Code and Ada Listing Files" on page 51.

You may be able to disregard other switch settings initially, depending on the requirements of your RCI extension.

Table 3-1 Rci Session and Library Switches

Name	Type	Default	Function
Session_Rci.Auto_Create_Remote_Directory	Boolean	True	If True, when you call a CMVC command to create an RCI view, the RCI automatically creates the remote directory for the view. See "Controlling Remote-Directory Creation" on page 50.
Rci.Auto_Transfer	Boolean	False	If True, the RCI automatically transfers units during the installed to coded phase in batch mode. If False, units must be transferred during the Build_Script operation by specifying the Transfer_To_Target parameter to be True. See "Controlling Batch Unit Transfers" on page 49.
Rci.Compiler_Post_Options	String	Null string	Contains a string value to append to the unit name in the target-compiler command.
Rci.Compiler_Pre_Options	String	Null string	Contains a string value to insert before the unit name in the target-compiler command.
Rci.Host_Only	Boolean	False	If True, the RCI compiles the RCI view but does not download and compile the units remotely. See "Turning Off Remote Compilation" on page 49.
Rci.Linker_Post_Options	String	Null string	Contains a string value to append to the unit name in the remote linker command.
Rci.Linker_Pre_Options	String	Null string	Contains a string value to insert before the unit name in the remote linker command.
Rci.Operation_Mode	String	Null string	Contains the values "interactive" or "batch." This switch overrides the default operation mode for compilation within the view. A value of null string or any invalid value causes RCI to use the default operation mode, determined by the customization or the Target_Key.Register command. See "Choosing Interactive or Batch Operations" on page 49.
Rci.Optimize_Download	Boolean	True	If False, the RCI downloads units to the remote compilation platform, if coding in interactive mode, regardless of whether their edit times have changed since the last time they were coded.
Rci.Remote_Library	String	Null string	Contains the name of the remote program library, stored as state information. *Do not alter this value.*

Table 3-1 Rci Session and Library Switches (continued)

Name	Type	Default	Function
Session_Rci.Retrieve_Executable	Boolean	False	Controls whether the RCI uploads the executable file created by the remote linker. If True, the RCI checks any extension switches that control uploading executable files. This action depends on your extension. See your extension user's guide or your customizer for more information.
Rci.Trace_Command_Output	Boolean	False	If True, the RCI displays the text of remote commands executed during the coding and linking steps. See "Displaying Remote Process Commands" on page 49 and "Output from the RCI Compiler and Linker" on page 56. It may display other information also, depending on the extension.

Table 3-2 Rci and Session_Rci Custom Switches

Name	Type	Default	Function
Rci.*Custom*_Default_Machine	String	Null string	Specifies the remote compilation platform for newly created views (one per registered target key per library).
Rci.*Custom*_Default_Roof	String	Null string	Specifies the roof value used to construct a remote directory name for newly created views (one per registered target key per library).
Session_Rci.*Custom*_Default_Machine	String	Null string	Specifies the remote compilation platform for newly created views (one per registered target key per session).
Session_Rci.*Custom*_Default_Roof	String	Null string	Specifies the roof value used to construct a remote directory name for newly created views (one per registered target key per session).

Table 3-3 Sample Rci Switches

Name	Type	Default	Function
*Custom*_Assemble	Boolean	False	If True, sets the `-a` option for the target compiler, which creates the `.s` file on the compilation platform and the `.<Asm>` file on the host.
*Custom*_List	Boolean	False	If True, sets the `-l` option for the target compiler, which produces an Ada listing file with the `.lst` extension on the compilation platform and the `.<List>` file on the host.
*Custom*_Suppress_All	Boolean	False	If True, sets the `-s` option for the target compiler, which has the same effect as pragma Suppress.
*Custom*_Verbose	Boolean	False	If True, sets the `-v` option for the target compiler, which displays informational messages during the compilation that normally are not displayed.

Viewing Switches

These switches can be viewed in the following ways:

- Switches.Edit displays all library switches associated with the view, including an asterisk to indicate nondefault values.
- Switches.Edit_Session_Attributes displays the user's session switches.
- Switches.Display displays any switches associated with the library that have nondefault values; in addition, if any Rci switch has a nondefault value, all Rci switches are displayed.
- Rci.Show_Remote_Information displays the values of Ftp.Remote_Directory and Ftp.Remote_Machine.

Figure 3-3 shows a sample listing of the current switch values when viewed using the Switches.Display command.

```
!Users.Rh.Planetary_Motion_Rci.Rev1_Rs6000_Aix_Ibm_Working.Units :   Library (Di
        Ftp . Account                                  : String         := ""
        Ftp . Auto_Login                               : Boolean        := False
        Rci . Auto_Transfer                            : Boolean        := False
        Rci . Compiler_Post_Options                    : String         := ""
        Rci . Compiler_Pre_Options                     : String         := ""
        Rci . Host_Only                                : Boolean        := False
        Rci . Linker_Post_Options                      : String         := ""
        Rci . Linker_Pre_Options                       : String         := ""
        Rci . Operation_Mode                           : String         := ""
        Rci . Optimize_Download                        : Boolean        := True
        Ftp . Password                                 : String         := ""
        Ftp . Prompt_For_Account                       : Boolean        := False
        Ftp . Prompt_For_Password                      : Boolean        := False
        Ftp . Remote_Directory                         : String         := ""
        Rci . Remote_Library                           : String         := ""
        Ftp . Remote_Machine                           : String         := ""
        Ftp . Remote_Roof                              : String         := ""
        Ftp . Remote_Type                              : String         := ""
        Ftp . Send_Port_Enabled                        : Boolean        := True
        Rci . Trace_Command_Output                     : Boolean        := False
        Ftp . Transfer_Mode                            : Mode_Code      := Nil
        Ftp . Transfer_Structure                       : Structure_Code := Nil
        Ftp . Transfer_Type                            : Type_Code      := Nil
        Ftp . Username                                 : String         := ""
        Rci . Vax_Vms_Dec_Xt_Brief                     : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Check                     : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Cross_Ref                 : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Debug                     : String         := ""
        Rci . Vax_Vms_Dec_Xt_Default_Machine           : String         := ""
        Rci . Vax_Vms_Dec_Xt_Default_Roof              : String         := ""
        Rci . Vax_Vms_Dec_Xt_Full                      : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Ldebug                    : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_List                      : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Map                       : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Nocheck                   : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Nodebug                   : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Noexecutable              : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Nomain                    : Boolean        := False
        Rci . Vax_Vms_Dec_Xt_Optimize                  : String         := "(NONE)"
        Rci . Vax_Vms_Dec_Xt_Show                      : String         := ""

= !USERS.RH.LIBRARY_SWITCHES "Rci" (switch)    (All Switches)
```

Figure 3-3 Switches Viewed Using Switches.Display

If the Rci switches do not appear when the Switches.Edit or Switches.Display command is executed, the RCI may not be running on your machine. See "Verifying the RCI Installation" on page 18 in Chapter 2.

Changing Switch Values

Switch values can be altered in the following ways:

- From the display produced by Switches.Edit or Switches.Edit_Session_Attributes, position the cursor on the line containing the switch to be changed and press [Edit]. Fill in an appropriate value and press [Promote]. The displayed value is changed but the new value does not take effect until Common.Commit ([Enter] or [Commit]) or Common.Promote ([Promote]) is used on the switch window.
- With the Switches.Set command for each switch to be altered, as in:

```
Switches.Set ("Ftp.Username=""frederick""");
```

Turning Off Remote Compilation

Set the Host_Only library switch to True when you do not want an RCI view compiled on the remote compilation platform. Views controlled by Host_Only => True provide specs for semantic checking, but when units in a Host_Only view are coded in the Environment, they are not transferred and compiled remotely.

This feature is particularly useful when you have a large piece of common code that changes infrequently and already resides in a remote library.

Use the Host_Only switch in combination with the Trace_Command_Output switch as you develop code in the host environment. For example, from a host-only view, you can check the commands that the RCI would send to the remote compilation platform without actually downloading and compiling on the target compiler.

Displaying Remote Process Commands

Set the Trace_Command_Output switch to True when you want to display the text of commands sent to the remote machine. This can be useful in testing compile, link, and process commands sent to the remote machine and in gathering troubleshooting information about RCI remote commands that fail.

You can use this switch in combination with the Host_Only switch to gather information about what is sent to the remote machine. For example, from a host-only view, you can check the series of commands that the RCI would send to the remote machine without downloading and compiling on the target compiler.

Choosing Interactive or Batch Operations

The RCI handles remote compilation in one of two modes: interactive or batch. Use the Rci.Operation_Mode switch to select the mode on a view-by-view basis. The switch overrides the default value set in the extension or set by the *Custom_Key*-.Register command for the view that contains the switch.

Set the switch to a string value of either "interactive" or "batch". A null string or an invalid value causes the RCI to use the default operation mode.

Controlling Batch Unit Transfers

The Rci.Auto_Transfer switch determines whether the RCI automatically transfers units to the remote compilation platform when you promote those units from in-

stalled to coded on the host. The default value is False. Set the value of the switch to True if you want units to transfer automatically. You can also override the default value for a single batch script by setting the Transfer_To_Target parameter of the Rci.Build_Script command to True.

Controlling Remote-Directory Creation

When you use a CMVC command to create an RCI view, the RCI checks the value of the Session_Rci.Auto_Create_Remote_Directory switch to determine whether to create the corresponding remote directory on the compilation platform at the same time it creates the new host view. The default value for the switch is True. If you do not want to create a remote directory when the RCI creates the host view, set the switch to False.

Setting Switches for Remote Communication

For the RCI to be able to download and compile an Ada unit when promoting it to the coded state, the RCI must be able to establish communication with the remote machine, as described in "Specifying Remote Login Information" on page 20 in Chapter 2 and "Changing Switch Values," above. A common solution is to use the Ftp switches described in Table 3-4 and located in the switch file that is associated with the view in which the Ada unit is located. All of these switches are of type String and default to null. The RCI sets the values for Remote_Machine and Remote_Directory when the view is created from the default switch-naming scheme or command parameter (see the Initial and Make_Path commands for package Cmvc).

Table 3-4 Ftp Switches

Name	Function
Password	Specifies the password for the remote user.
Remote_Directory	Specifies the remote directory for downloading and compiling.
Remote_Machine	Specifies the remote machine with which communications should take place as defined by the Transport_Name_Map file.
Username	Specifies the username for the requested remote machine.

Additional information about Ftp switches can be found in the File Transfer Protocol (FTP) book of the *Rational Networking—TCP/IP Reference Manual.*

Setting Switches for Target-Compiler Operations

The Rci switches whose names begin with *Custom* affect the way in which the target compiler operates; they map directly to target-compiler options as suggested in Figure 3-3. These switches remain enabled until the target key is unregistered; at next registration, they return to the default values for the customization. For more information on target options, refer to the manual for the target compiler and the user's guide for your customization.

The default values of these RCI extension switches serve for the general RCI case. Of these switches, the extension is likely to have switches similar to the following, which affect the operation or output of the RCI as described in this manual:

- *Custom*_Assemble and *Custom*_List: See "Saving Assembly Source Code and Ada Listing Files," below.
- *Custom*_Verbose: Affects output from the target compiler and remote linker.

Saving Assembly Source Code and Ada Listing Files

If provided for by the target compiler and by the RCI customizer, certain switches specify whether associated files for the Ada units should be created on the host as described in "Output from the RCI Compiler and Linker" on page 56.

To save the assembly-language source listing generated by the target compiler and transfer it into an associated file on the host, set the *Custom*_Assemble-equivalent switch to True. During the interactive coding step, the assembly-language listing file is created on the compilation platform and transferred to the host into an associated file of the original Ada unit, usually named .<Asm> (see Table 3-6, "Sample Suffixes Identifying Associated Files on the R1000," on page 58).

To save the Ada source listing generated by the target compiler, set the *Custom*_List-equivalent switch to True. During the coding step, the Ada source listing file will be created on the compilation platform and transferred to the host into an associated file of the original Ada unit, usually named .<List>.

In batch mode, the target compiler produces these files on the compilation platform, but they are not automatically uploaded to the host. To transfer them back to the host, use the Rci.Upload_Associated_Files command described in "Retrieving Associated Files" on page 72.

Specifying Unit-Specific Compiler Options

The RCI allows you to modify compiler options on a unit-by-unit basis. Use the Rci.Set_Unit_Option command to change the compiler options for a particular unit. When the unit is compiled, these values are used for compiler-option values instead of the current values in the library-switch file. The RCI stores the new value set by the Rci.Set_Unit_Option command with the state information. This value remains enabled until you undo it with the Rci.Remove_Unit_Option command.

To set the value of a specific option for a given unit:

1. Enter the Rci.Set_Unit_Option command and press **[Complete]**:

```
List of possible completions
SET_UNIT_OPTION =>
procedure Set_Unit_Option
             (Option_Switch : String := ">>OPTION<<";
              Switch_Value  : Boolean;
              Units         : String := "<CURSOR>";
              Response      : String := "<PROFILE>");
procedure Set_Unit_Option
             (Option_Switch : String := ">>OPTION<<";
              Switch_Value  : String;
              Units         : String := "<CURSOR>";
              Response      : String := "<PROFILE>");
```

2. Choose the appropriate command to modify your compiler option. If the compiler option does not have arguments, select the first completion; if it does have arguments, select the second completion.

Put the cursor on the line with the appropriate command, select the command, and press [Complete] to copy that command into the command window:

```
Rci.Set_Unit_Option (Option_Switch => ">>OPTION<<",
                     Switch_Value  => True or ""
                     Units         => "<CURSOR>",
                     Response      => "<PROFILE>");
```

3. Fill in the parameters as follows:
   - Option_Switch: Fill in the name of the compiler option in the RCI view library- switch file. This can be the name of the compiler option as it appears in the switch—for example, Optimize—or the fully qualified switch name of the library switch, Rci.I186_Vms_Ddc_Optimize.
   - Switch_Value:
     - For compiler options with no arguments, set the value to True to enable the compiler switch or False to disable it.
     - For compiler options with arguments, fill in the argument's desired value. An ampersand means that the option is present with no argument, and an empty string means the option is not present.
   - Units: Specify the units for which the compiler-option values apply.
4. Press [Promote]. The compiler-option values are assigned for the specified units.

For example, to set the compiler option Optimize to use the argument value Time for the unit indicated at the cursor, enter the following command:

```
Rci.Set_Unit_Option
             (Option_Switch => "Rci.I186_Vms_Ddc_Optimize",
              Switch_Value  => "Time",
              Units         => "<CURSOR>",
              Response      => "<PROFILE>");
```

Rci.Display_Unit_Options displays the options set by Rci.Set_Unit_Option for the specified units.

Values set with the Rci.Set_Unit_Option commands remain enabled, regardless of changes to the library-switch file, until you remove them with Rci.Remove_Unit_Option. After you have removed the set option, the compiler option's value is taken from the library-switch file for the view.

CREATING AN ADA PROGRAM FOR REMOTE COMPILATION

When you create Ada programs for remote compilation, you need to consider differences in the way the remote compilation system handles target-dependent elements of the programs. If you have not already done so, you may want to read "Comparing Features Of Rational Environment And Rci Compilation" on page 7 in Chapter 1. Specifically, regarding the creation of Ada code for the remote machine, you need to be aware that the remote compilation system handles generics and inlined subprograms, packed records and arrays, record representations, and implementation-dependent pragmas differently than does the R1000 compilation system.

Creating an Ada Main Unit

When an Ada main unit is created under the RCI to run on the compilation platform, the process is the same as when an Ada unit is created to run on the R1000.

1. Go to the Units directory of the view in which the unit should be created.
2. Press [Object] - [I] to create an open window ready for editing.
3. Enter text as shown in Figure 3-4. The use of pragma Main may be required or ignored by the target compiler; it may exist in a joined unit in an R1000 view to affect behavior on the host, and its presence may or may not affect remote compilation (although a warning may be issued).

```
with Test_Ada_Io;

procedure Test_Ada is
begin
    Test_Ada_Io.Request_Response;
end Test_Ada;

pragma Main;
```

Figure 3-4 Ada Unit with Pragma Main

Restrictions on a main unit for your extension are described in the manual for the target compiler.

Before the main unit can be promoted to the coded state, all units in the *with* closure of the unit must be in the coded state.

Note: *Consider giving the unit a name whose first customization-specified number of characters (or fewer; see Appendix C) are unique compared to other units. The host unit name might be truncated to create a text filename when the unit is downloaded to the compilation platform. Creating these names for remote units is described in* *"Remote Files and Names" on page 56.*

Using Pragma Main

The RCI supports the Rational-defined pragma Main. If you specify pragma Main in the specification of a main program, the RCI invokes the remote link operation automatically after coding the unit. This produces an executable module without explicitly invoking the Rci.Link command. Since this eliminates the need to explicitly link the main program, the RCI development model more closely matches the Rational Environment native development model. RCI support of this feature is particularly useful when you do host-based testing and make the transition from CDF to RCI use.

To use pragma Main, place it immediately after the end of the specification or body of a main procedure. You can use pragma Main in any library-level Ada unit that can be linked on the remote compilation platform.

Before a main program can be promoted to the coded state, all compilation units in its closure must be in the coded state. If all units in the closure are not coded, the RCI creates a message of the following form:

```
ERROR Link failed for <Main_Unit>.  Failure in Host_Linker.Prelink.
```

Attempting to demote a unit in the closure of a coded main unit can result in an error message.

RCI support for pragma Main does not include parameters, which means that the RCI issues warning messages in code joined with R1000 or CDF views where pragma Main can include various parameters.

Refer to Appendix F of the Ada LRM for more information about the R1000 semantics of pragma Main.

Using Pragma Inline

The RCI supports pragma Inline. It takes into account inlining dependencies when computing coding order during promotion and obsolescence requirements during demotion. Units are compiled in an order that allows the target compiler to inline (macro expand) calls to inlined routines. Inlined callers are obsolesced when the body of an inlined routine is demoted.

Note that inlining is always in effect, regardless of how your site-specific customization extension or Rational-supported extension has been implemented.

Adding pragma Inline incrementally to a unit is prohibited and produces a message.

Using Representation Clauses

The target compiler may place restrictions on representation clauses, which the RCI enforces. Your customizer determines these restrictions based on the restrictions of the target compiler, and the RCI enforces the rules specified for your extension. The customizer can provide a list of restrictions for length, record, and address clauses in Appendix C of this manual.

Note: *Only restrictions specified by the customizer are caught during the promotion to the installed state; limitations imposed by the target compiler but not by the customization are not caught until remote compilation occurs.*

Using Implementation-Dependent Pragmas

The target compiler may support pragmas not found in the native compiler; the RCI supports all LRM standard pragmas as well as the target compiler's target-dependent pragmas specified by the RCI customization. Some possible target pragmas are described in Table 3-5. For further information on target-compiler pragmas, see the manual for the target compiler and the Ada LRM. Appendix C provides a blank table for the customizer to fill in valid pragmas.

See the discussion of pragma Main in "Creating an Ada Main Unit," above.

Table 3-5 Sample Target Pragmas Recognized by RCI

Pragma	Parameters	Function
Comment	*String_Literal*	
Images	*Enumeration_Type, When_Generated*	Controls the generation of image tables for enumeration types. When_Generated can be either Immediate or Deferred.
Linkname	*Interfaced_Subprogram-_Name, Link_Name*	Allows pragma Interface to work with routines whose names are nonstandard.
Os_Task	*Priority*	Directs the placement of tasks into processes under the remote operating system.

REMOTELY COMPILING AND LINKING A SIMPLE ADA PROGRAM

In the interactive mode of the RCI, the act of promoting a unit to the coded state performs a number of operations automatically and may create several output files. The same is true for the linking step, which is separate from the coding step. This section describes these steps and output.

Creating an Executable Program

If your target platform is the same as your remote compilation platform, your target compiler is a native compiler. If your target platform is different from your compilation platform, your target compiler is a cross-compiler. In either case, the target compiler generates code for the target platform.

To create a program that is executable on the remote machine from existing Ada source code:

1. Select the Ada unit that you have created on the host in an RCI view.
2. Press [Promote] to promote from the source state to the installed state. See "What Happens During the Installing Step" on page 58 for additional information.
3. Press [Promote] again to promote from the installed state to the coded state. As described in "What Happens During the Coding Step" on page 59, the unit is downloaded to the remote machine and compiled with the remote compiler. Various output may be displayed and files created as described in the "Output from the RCI Compiler and Linker," below.

   Note: *If you need to compile more than one unit or to make sure that the entire closure of a given main unit is in the coded state, you can use the* [Code (This World)] *key or the Compilation.Make command.*
4. Enter the Rci.Link command in the command window associated with the Ada *main* unit that includes the newly coded unit in its closure and press [Promote].

The unit on the remote machine is linked as described in the "What Happens During the Linking Step" on page 61.

To execute and debug the program, log onto the remote machine and use the remote operating-system utilities or use remote commands from the host to transfer and control the target execution.

Output from the RCI Compiler and Linker

When an Ada unit is promoted to the coded state or linked, various output can be displayed in a host window, and certain files associated with the unit may be produced automatically on the R1000 and on the compilation platform, as shown in Figure 3-6 on page 60 and Figure 3-7 on page 62.

Displaying Remote Commands

If the Rci.Trace_Command_Output switch is True, then the actual remote command used to compile or link a file is displayed in the response window. It may be beneficial to display this if several RCI switches are used to specify nondefault target-compiler options.

Displaying Remote Standard Output

Standard output from the execution of a remote command is always displayed in the current output window. Switches like the Rci.*Custom*_Verbose switch may provide additional informational messages.

Remote Files and Names

Files created on the compilation platform are determined by Rci switch settings and compiler commands. See "Setting Session and Library Switches" on page 45.

Remote filenames are stored permanently on the host for each unit and are calculated from the simple name of the host unit. For example, names for the RS6000 platform are formed as follows:

- A suffix consisting of an underscore and a serial number is appended to the name based on the order in which the unit was registered in the RCI state information. This guarantees that each remote name is unique.
- An additional suffix of _b is appended for an Ada body and _s for an Ada specification.
- The .ada extension is appended to the result for Ada bodies and specifications that the RCI downloads. The target compiler may create files with other extensions as described below.
- Secondary text files are downloaded to the name specified in the Rci.Create_Secondary command or by default with the same name as on the host.
- The simple name before the suffixes is truncated so that the entire name is no longer than the customization-specified number of characters (see Appendix C or the user's guide for your extension).

Note: *The Rci.Set_Remote_Unit_Name command can be used to override this default name, but it is then the user's responsibility to verify that no duplicate names are specified.*

Logging onto the remote machine and using a command from the remote operating system to view the files in the remote directory might reveal a listing like this:

```
planetary_motion
planetary_motion_1_s.ada
planetary_motion_1_s.lst
planetary_motion_1_s.s
planetary_motion_2_b.ada
planetary_motion_2_b.lst
planetary_motion_2_b.s
screen_io_c
the_file_with_an_extremely_long_3_s.ada
```

In this list, the .ada files are Ada units that were downloaded from the host; the screen_io_c is a secondary text file whose original host name before downloading was screen_io_c; the .lst files were created by the target compiler and contain Ada listing files; the .s files were also created by the compiler and contain assembly-language listings. The file with the long name was called The_File_With_An_Extremely_Long_Name on the host. The file with no extension is the executable module.

In addition, there might be entries for the default remote program library and the remote import list. See Figures 3-6 and 3-7 and Appendix C for more information.

Host Associated Files

Promoting a host unit to the coded state may result in the creation of *associated files* on the host, such as Planetary_Motion.<List>, as shown in the example of a directory in Figure 3-5.

```
!Users.Rh.Planetary_Motion_Rci.Rev1_Rs6000_Aix_Ibm_Working.Units :   Library (Di
  Input_Operations : C Ada (Pack_Spec);
    .<Asm>         :   File;
    .<List>        :   File;
  Input_Operations : C Ada (Pack_Body);
    .<Asm>         :   File;
    .<List>        :   File;
  Orbit            : C Ada (Pack_Spec);
    .<Asm>         :   File;
    .<List>        :   File;
  Orbit            : C Ada (Pack_Body);
    .<Asm>         :   File;
    .<List>        :   File;
  Planetary_Motion : C Ada (Main_Proc)
    .<Asm>         :   File;
    .<List>        :   File;
    .<exe>         :   File (Binary);
  Planetary_Motion : C Ada (Main_Body);
    .<Asm>         :   File;
    .<List>        :   File;
```

Figure 3-5 Directory Showing Associated Files After Coding and Linking

Associated files are automatically created and/or retained if the appropriate library switches are set to True, as suggested in Table 3-6. See Figures 3-6 and 3-7 for more information. Actual associated filenames and switch settings are determined by the customization; a table in which the customizer can list these values is provided in Appendix C.

Table 3-6 Sample Suffixes Identifying Associated Files on the R1000

Suffix	Sample Description of File
.<Asm>	Listing of the assembly-language source. The file's creation is controlled by the Rci.*Custom*_Assemble switch. The file is created on the compilation platform and uploaded to the host as a text file.
.<List>	Ada source listing output by the target compiler. The file's creation is controlled by the Rci.*Custom*_List switch. The file is created on the compilation platform and uploaded to the host as a text file.
.<Exe>	Executable module in target object-module format. The file is created by the remote linker and is uploaded to the host (if Session_Rci.Retrieve_Executable is True) during execution of the Rci.Link command. The file is not a text file.

The source listing and assembly-language files can be useful for debugging, and the <Exe> file can be useful for configuration-management purposes or for direct final downloading to the target.

If the parent unit is deleted, all of its associated files are deleted automatically. If you need permanent copies of these files, you can make and control nonassociated files from them. These files cannot be produced directly; they result only from invoking the RCI target compiler and RCI remote linker.

The associated files of an Ada unit on the host have names formed by adding a suffix to the simple name of the unit. Because the suffixed name is enclosed in angle brackets, these files are often referred to as *pointy files*.

What Happens During the Installing Step

When an Ada unit is promoted to the installed state in an RCI view, either with the [Promote] key or with the command:

```
Compilation.Make (Unit => "My_Unit",
                  Goal => Compilation.Installed);
```

the RCI performs a number of operations, including target-dependent and target-independent semantic checking. This includes:

- Verifying the correctness of compiler-specific representation clauses and attributes.
- Checking the validity of LRM-predefined and implementation-defined pragmas (those allowed by the target compiler). In checking implementation-defined pragmas, the RCI verifies the following:
  - Is the pragma name valid?
  - Does the pragma appear in a valid context?
  - Are the types of the pragma's arguments valid?
  - Do the arguments have legal values?
  - Is the quantity and ordering of the arguments correct?
- Checking references to the target-compiler-predefined types, objects, and routines and comparing them with the specifications for all target-compiler-predefined units, including Standard, System, and Text_Io, in !Targets.*Custom_Key*.

The RCI assumes that a unit is semantically correct with respect to the target compiler once it has been installed. However, changes to the target compiler or features of the compiler that are not included in the RCI extension for reasons such as being difficult to model on the R1000 may result in semantic errors when the unit is compiled remotely.

In addition, information about the unit is made known to *(registered with)* the RCI state information as described in "Rci State Information" on page 94 of Chapter 6.

See the *Rational Environment Reference Manual* for further information on what occurs during the installing step.

What Happens During the Coding Step

When an Ada unit is promoted from the installed state to the coded state in an RCI view, either with the [Promote] key or with the command:

```
Compilation.Make (Unit => "My_Unit",
                  Goal => Compilation.Coded);
```

the RCI, in interactive mode, performs these steps:

1. The RCI checks whether the units in the closure of the requested unit(s) are in the coded state. If not, as in the native compiler, the system tries to automatically code all the units so long as those units are already in the installed state. Compilation ordering is based on standard Ada dependencies and implementation-dependent dependencies such as those imposed by inlining and generic macro expansion.
2. The first unit in the compilation order, if its edit time on the host is more recent that its latest download time, is downloaded to the compilation platform using the remote connection information described in "Setting Up Remote Communications" in Chapter 2. Naming of the remote units is discussed in "Remote Files and Names" on page 56. (See "Processing Secondaries" on page 105 in Chapter 7.)

   If the connection fails, an error message is displayed and the unit is not promoted to the coded state.
3. The RCI issues a command to the target compiler for that unit. The command is constructed as follows:
   - If the unit has a secondary (described in Chapter 7, "Using Non-Ada Code with the RCI"), the command that is associated with the unit (see the Rci.Show_Secondary command) when the secondary was created or the Rci.Set_Secondary_Command is the command used by the remote operating system to process the secondary unit. Rci switches are not used.
   - Otherwise, the options determined by the Rci.*Custom* switches are used with the default compiler command (see Appendix C) to construct the command for the target compiler.

   The RCI waits for the compilation to complete before progressing to the next step. Standard output from the target compiler, including errors, is displayed according to the response profile. If errors occur, the RCI compiler terminates without requesting compilation for any other units. Among the errors that can prevent a unit from being promoted to the coded state are:
   - Network errors
   - Invalid remote username and password

- Unknown remote machine or directory names
- Existence of nonterminals in the code (statement prompts and so on, as described in the *Rational Environment Reference Manual*)
- Semantic errors caught by the target compiler that were not caught during the promote-to-installed step

4. When a unit is successfully compiled by the target compiler, its state is changed to coded and its consistency information is updated in the RCI state information.
5. If the Rci.*Custom*_Assemble-equivalent or Rci.*Custom*_List-equivalent switches are set, the appropriate assembly-language source listing and Ada source listing files are uploaded to the host and saved in associated files as described in "Output from the RCI Compiler and Linker" on page 56.
6. The previous four steps are repeated for each unit that needs compilation.

The program is now ready to be linked.

Note: *Promoting to the coded state does not change the executable module on the compilation platform; whenever any unit is changed, the main unit must be relinked.*

See the *Rational Environment Reference Manual* for further information on what happens during the coding step.

Input to and output from the compilation step is shown in Figure 3-6. Note that file suffixes may differ, depending on the extension.

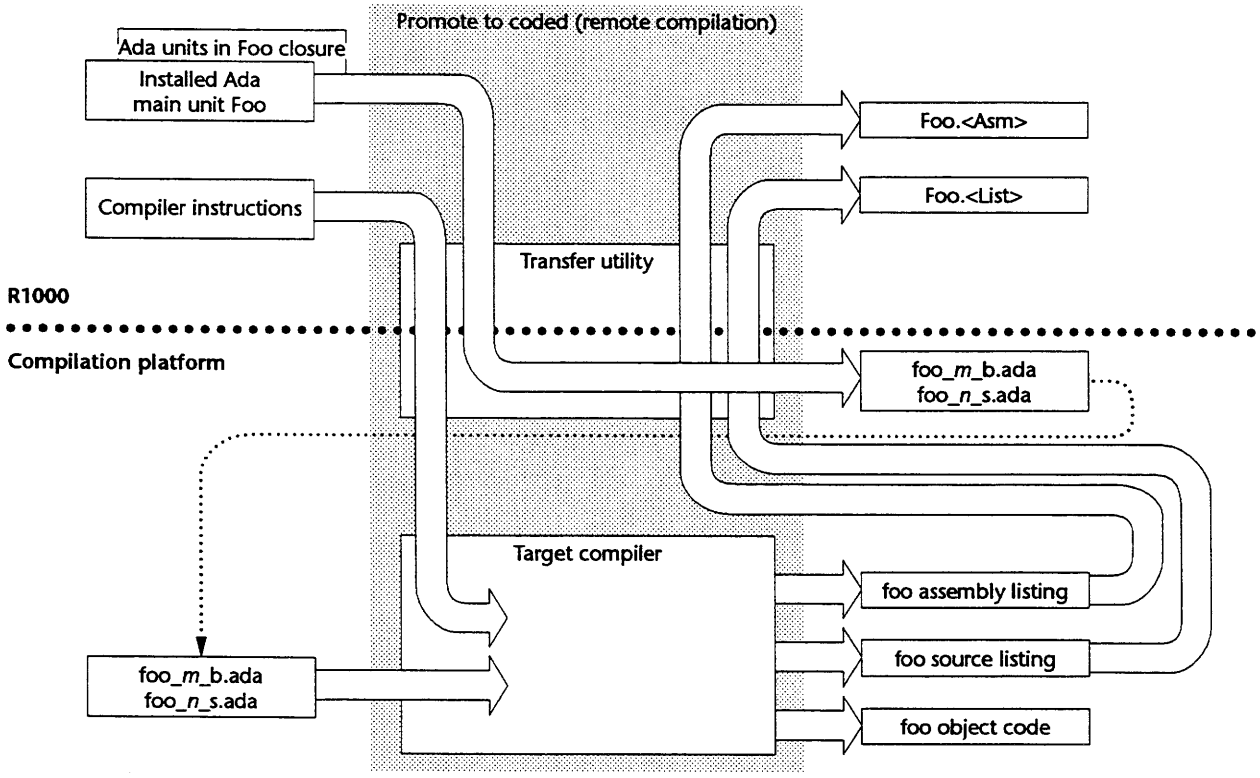

Figure 3-6 Input to and Output from the RCI Compiler

The RCI takes different steps in batch operation mode. See Chapter 4, "Using Batch Processing with the RCI," for a description of batch coding.

What Happens During the Linking Step

Promoting an Ada unit from the installed state to the coded state causes it to be compiled, but not linked, on the compilation platform. When the Rci.Link procedure is invoked, the RCI follows these steps:

1. The RCI checks whether all units in the closure of the requested main unit are in the coded state. If not, the RCI linker uses the value of the Make_Uncoded_Units parameter to decide whether to attempt to code the necessary units.

   If Make_Uncoded_Units is True, it invokes the RCI compiler on each unit as described in the previous subsection.

   If Make_Uncoded_Units is False and a necessary unit is not coded, or if any attempted compilations fail, the RCI displays a message and the link fails.
2. A connection to the remote machine is made using the machine, directory, username and password as described in "Setting Up Remote Communications" on page 19 in Chapter 2.
3. The remote linker is invoked, and instructions are taken from the default linker command (see Appendix C). The appropriate options from the Rci switch file are concatenated to the link command.
4. Standard output from the remote linker is displayed according to the response profile. If any errors occur, a new executable module is not generated on the compilation platform (and hence not uploaded to the host). Errors are generated if the Ada unit does not qualify as a main unit.
5. After a successful link, the linked executable module is saved on the compilation platform with the name specified in the Executable_Name parameter of the Link command. If Executable_Name is null, the name is the same name as the simple name of the host Ada unit from which the executable module was generated; the extension .<Exe> is appended to the name if the target compiler allows such extensions.

In addition, if the Session_Rci.Retrieve_Executable switch is set to True, the linked executable module is uploaded to the host and saved as an associated file of the Ada unit, typically named .<Exe>. This module is not executable on the R1000 host, but it can be controlled and managed by taking advantage of CMVC and RCI facilities for cross-system consistency management. It could also be downloaded directly to a target from the R1000 using FTP utilities.

Input to and output from the Rci.Link command is shown in Figure 3-7 (note that the associated filename may differ). See also "Output from the RCI Compiler and Linker" on page 56.

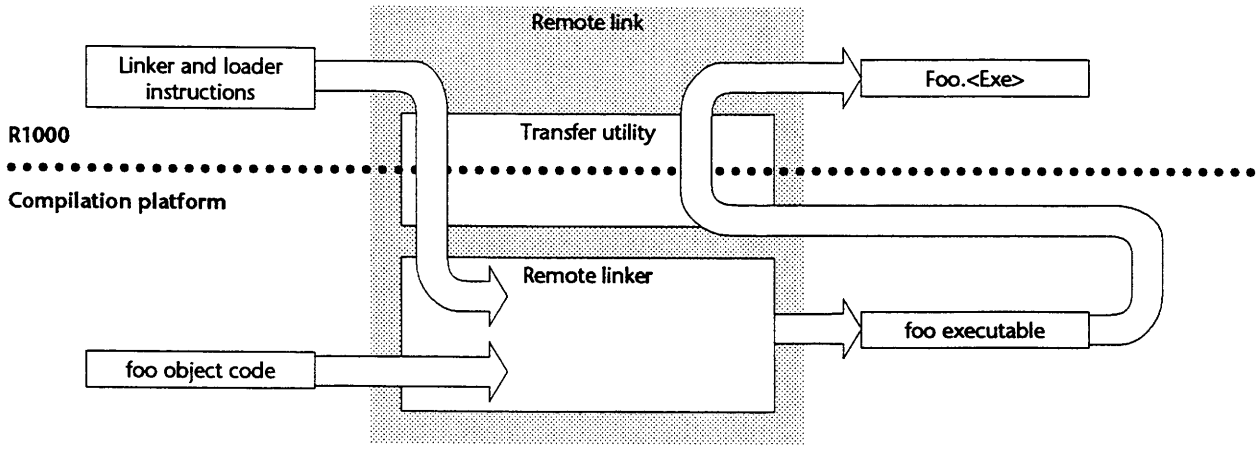

Figure 3-7 Input to and Output from RciLink

DEMOTING A UNIT

Either the [Demote] key or the Compilation.Demote command can be used to demote a coded unit to the installed state and from there to the source state. This process and standard coding dependencies are described in more detail in the *Rational Environment Reference Manual.* Different actions are taken depending on the particular customization. Actions performed by the RCI that are different from the standard Environment actions include:

- Demoting a generic body from coded to installed causes all instantiations of that generic to become obsolete and to be demoted to the installed state.
- Demoting inlined procedures causes units that call the inlined procedure to be demoted.
- Demoting any unit with associated files deletes the associated files.

Note: *Demoting a unit from coded to installed or source has no impact on the state of units in the remote program library.*

No action is taken on the compilation platform.

4 Using Batch Processing with the RCI

This chapter addresses how to use the RCI in *batch mode* to perform batch processing. Under batch mode, promote and demote operations affect only units on the host; when you are ready to compile units on the remote compilation platform, you use batch operations to download units, build and download a batch compilation script, and execute that script.

This chapter includes the following sections:

- Overview of batch mode
- Preparing to use batch mode
- Using batch-mode operations
- Troubleshooting batch-mode operations

OVERVIEW OF BATCH MODE

The Environment's compilation model is optimized to support quick turnaround of changes. For example, Environment users can make changes incrementally for the best turnaround. Although it is possible to issue batch-style compilation commands, the Environment does not optimize for this case.

The RCI provides added flexibility for compilation by allowing you to work either in interactive mode or in batch mode. Each mode has its advantages—whereas interactive mode maintains continuous consistency between host and remote units, batch mode decouples host and remote compilation and reduces network traffic.

The following subsections describe:

- The differences between interactive and batch modes with respect to compilation, associated-file retrieval, library management, and consistency management
- Common reasons for using batch mode
- Considerations when mixing batch and interactive modes

Compilation and Associated-File Retrieval

When you use the RCI in interactive mode, promoting a set of units to the coded state on the host causes each unit to be downloaded to and then compiled on the remote compilation platform, starting with the first unit in the compilation order. When a unit is successfully compiled by the target compiler, its state on the host is changed to coded; consequently, the coded state indicates that a unit is compiled both on the host and on the remote platform. Furthermore, associated files such as object files and listings are automatically uploaded to the host.

When you use the RCI in batch mode, promoting a set of units to the coded state on the host simply determines whether a proper compilation order can be generated for the units and their supplier and, if so, marks each unit as "codable" by changing its unit state to coded and entering a compilation timestamp for it in an internal database on the host. Although the units optionally can be downloaded as part of the promote operation (if the Rci.Auto_Transfer switch is set to True), these units are not compiled on the remote platform. Consequently, units on the host can be demoted and repromoted with no impact on remote compilation.

When you want to compile units on the remote platform, you promote the units to the coded state on the host and then use RCI batch operations to:

- Generate a *batch script* on the host (a batch script is a file containing remote commands that invoke the target compiler for the units requiring compilation)
- Download the units to the remote platform (if this was not done during the promote operation)
- Download and execute the batch script
- Upload the resulting associated files to the host

The RCI allows you to combine these operations or perform them as separate steps. For example, you can:

- Download units each time you promote them and then use a single command (Rci.Build_Script) to generate the batch script on the host and to download and execute the script on the remote compilation platform.
- Promote units without downloading them and then use the Rci.Build_Script command to generate the batch script, download the units, and download and execute the batch script.
- Promote units without downloading them and use the Rci.Build_Script command just to generate the batch script. At a later time, you can use the Rci.Transfer_Units command to download units to one or more remote compilation platforms and then use the Rci.Execute_Script command to download and execute the batch script on these platforms.

In all three of these scenarios, you upload the files that result from remote compilation using the Rci.Upload_Associated_Files command.

Thus, the RCI in batch mode allows you considerable flexibility in your development process—you can promote, demote, and repromote units on the host without causing remote compilation; conversely, you can force the remote recompilation of coded host units without having to demote and repromote them.

The Batch Script

When you enter the Rci.Build_Script command to generate a batch script for a unit, this command verifies that the unit's execution closure is in the coded state on the host, determines which units require remote compilation, and determines the order in which to compile them. The command then generates a script that contains a target compiler command for each unit. In addition, the command enters a build timestamp in an internal database for each unit included in the script.

The Rci.Build_Script command generates an *incremental* batch script in that it includes only obsolete units in the script. Obsolete units are units that have been coded on the host since the last time they were entered in a batch script. More specifically, the Rci.Build_Script command examines each unit in the execution closure,

compares the unit's compilation timestamp to its build timestamp, and includes the unit in the script only if the compilation timestamp is the more recent.

The RCI provides a special batch extension to allow customizers to further optimize the batch script so that it makes the fewest possible calls to the target compiler. In this case, instead of generating a batch script that invokes the target compiler once for each Ada unit, a customized batch script should invoke the target compiler once for each set of units that belong to the same Ada program library.

The base RCI generates a batch script that contains only target compiler commands and secondary commands. The RCI provides special batch extensions to allow customizers to generate site-specific scripts. Such scripts may contain additional commands, such as MVS® commands and statements.

Note that customizers can use batch extensions to cause the RCI to parse the results of a batch script execution and to make units on the host obsolete.

Remote Library and Consistency Management

The RCI provides interactive commands that automatically create and destroy remote libraries when you create, release, and destroy views on the host. You can use these commands regardless of which mode is in effect, provided that the host and the remote compilation platform are connected through the network.

When you use RCI in batch mode, you must still use the interactive commands to manage remote libraries, because the base RCI does not provide special batch operations for library management. Note that the RCI library extension can be customized to generate batch-style library-management commands. Check with your customizer to see whether such commands are available at your site.

When you use the RCI in interactive mode, consistency between the host and remote libraries is maintained automatically, provided that all code development takes place on the host. Such consistency can be maintained because information about remote unit state is sent back to the host with each interactive command. Consequently, updating a unit's state to coded on the host guarantees that it has compiled successfully on the target.

When you use the RCI in batch mode, it is more difficult to maintain consistency between host and remote libraries. In batch mode, there may not always be an automated way to determine whether the compilation script executed successfully on the target. The RCI provides ways to resolve inconsistencies between the host and target. Refer to "Maintaining Consistency in Batch Mode" on page 79.

When to Use Batch Mode

You may find it preferable or even necessary to use batch mode, particularly in the following cases:

- Where the remote operating system (for example, DOS or MVS) does not support concurrent Telnet or FTP connections. (See "Building Batch Scripts for Tape Environments" on page 72.)
- When there is significant cost associated with reelaborating the remote compiler for each Ada unit to be compiled. RCI batch extensions allow customizers to define batch scripts that contain the fewest possible compiler invocations.

- When you need tight control of resources on both the host and the remote compilation platform, so that, for example, compilation jobs must be scheduled overnight on some systems. Submitting a batch request on the remote platform generally provides quicker turnaround and tighter control over the compilation platform's resources.
- When doing large system builds from scratch. Whereas interactive operation is more efficient during ongoing development (because of the immediate feedback and the opportunity for making incremental changes), large system builds are usually done more efficiently in batch mode.
- When your project already uses the Target Build Utility (TBU), using RCI batch mode can provide a smooth upgrade for TBU users.

Mixing Batch and Interactive Operations

You can switch between batch and interactive mode at any time during development. The customizer establishes one of these modes as the basic mode when he or she registers the RCI for a particular target key. Users can override the basic mode by registering the RCI in the other mode or by changing the value of the Rci.Operation_Mode switch for a particular view.

Switching between modes can give rise to a mixed set of units, in which some units were promoted to the coded state under interactive mode and other units were promoted to the coded state under batch mode. The difference between units coded under each mode is significant:

- Units coded interactively are guaranteed to be compiled on the remote platform.
- Units coded in batch mode are strictly speaking "codable" rather than "coded." The "codable" state guarantees only that a proper compilation order may be generated for those units and their suppliers. There is no implication that these units have been downloaded to and compiled on the remote platform.

When a mixture of coded units exists, both batch-mode and interactive operations may end up referencing a combination of coded and codable units. Note that:

- Units coded under interactive mode are always considered "codable" by batch-mode operations. Thus, in batch mode, you can successfully generate batch scripts that reference interactively coded units. (Building a batch script for interactively coded units is useful for rebuilding an application, either in the same set of remote libraries or in a new set.)
- Units made codable under batch mode may, but need not, be considered "coded" by interactive operations. In particular, interactive compilation operations consider such units to be coded only if a batch script for these units was previously generated and successfully executed. Thus, in interactive mode, you can successfully compile new units against codable units only if the codable units have already been compiled on the remote platform.

PREPARING TO USE BATCH MODE

Before you can use the RCI in batch mode, you must:

- Set appropriate values for two library switches.
- Put batch mode into effect.

Setting Switches That Control Batch Operations

The following library switches control, for a given view, whether units are downloaded by promote operations and which mode is in effect (see also "Setting Session and Library Switches" on page 45):

- Rci.Auto_Transfer: Boolean := False;

  Set this switch to True if you want units to be downloaded to the remote compilation platform whenever you promote them from the installed state to the coded mode under batch mode.

  Set this switch to False (the default) if you do not want units to be downloaded by promote operations. In this case, you must download the units either as part of the Rci.Build_Script command (set the Transfer_To_Target parameter to True) or as a separate step (enter the Rci.Transfer_Units command).

  This switch does not affect operations in interactive mode.
- Rci.Operation_Mode: String := ""

  Set this switch to the null value (the default) if you want to use the mode in which the RCI was registered.

  Set this switch to Batch to use batch mode in the current view, regardless of the registered mode. (When you set the switch to Batch, be sure to verify that the Rci.Auto_Transfer library switch has the value you want.)

  Set this switch to Interactive to use interactive mode in the current view, regardless of the registered mode.

Putting Batch Mode into Effect

The default mode for the base RCI is interactive. Any of the following operations put batch mode into effect:

- Your customizer sets the operation mode to Batch in the customization template.
- You execute *Custom_Key*.Register (Batch_Mode => True) to register the extension in batch mode *and* you leave the Rci.Operation_Mode library switch set to null ("") in the relevant views.
- You set the Rci.Operation_Mode library switch to Batch in the relevant libraries, regardless of how the extension has been registered or customized.

When batch mode is in effect, be sure to verify that the Rci.Auto_Transfer library switch has the value you want.

Verifying Batch Registration

The various batch-mode operations fail with errors unless batch mode is in effect. To see whether batch mode is in effect, inspect the Rci.Operation_Mode library switch for the view you are working in:

- If the switch has the value Batch, then batch mode is in effect.
- If the switch has the value Interactive, change it to Batch.
- If the switch has the null value (""), then batch mode is in effect only if your extension has been registered in batch mode.

To find out how your extension has been registered, enter the What.Users command and search for the *Custom_Key* string for your extension:

- If the extension is in batch mode, you will find this string in an entry like the following:
  - RCI_Rev2_0_0_*Custom_Key*_FTP_Batch
  - RCI_Rev2_0_0_*Custom_Key*_DTIA_Batch
- If the extension is in interactive mode, you will find this string in an entry that does not end with "_Batch":
  - RCI_Rev2_0_0_*Custom_Key*_FTP
  - RCI_Rev2_0_0_*Custom_Key*_DTIA

USING BATCH-MODE OPERATIONS

Building Batch Scripts

In batch mode, promoting units to the coded state optionally downloads the units to the remote platform and marks them as codable in the RCI state file ("codable" means that a proper compilation order may be generated for the units and their suppliers). You must then enter the Rci.Build_Script (or Rci.Build_Script_Via_Tape) command as a separate operation to build a batch script for these units. All appropriate library options, unit options, secondary files and commands are included in the compilation script.

Before you can build a batch script for a given unit, its entire closure must be in the coded state. The closure may include units that were promoted to the coded state in either mode. In fact, building a batch script for interactively coded units is useful for rebuilding an application, either in the same set of remote libraries or in a new set.

The command you use to build a batch script depends on how the host and remote compilation platform communicate:

- You use the Rci.Build_Script command if communication is done through the network.
- You use the Rci.Build_Script_Via_Tape command if communication is done through tape. These are discussed in the following subsections.

Building Batch Scripts for Networked Environments

Use the Rci.Build_Script command if the host and the remote compilation platform communicate using Telnet and FTP:

1. Enter the Rci.Build_Script command and press [Complete]. The command looks like this:

```
Rci.Build_Script
   (Host_Units            => "<IMAGE>",
    Link_Main_Units       => True,
    Transfer_To_Target    => True,
    Host_Script_File      => "<DEFAULT>",
    Remote_Script_File    => ">> FULL REMOTE NAME <<",
    Build_List_File       => "<DEFAULT>",
    Execute_Script        => False,
```

```
Effort_Only           => False,
Minimal_Recompilation => True;
Make_Units            => False,
Response              => "<PROFILE>");
```

2. Specify the Host_Units parameter with the names of the units to be compiled and linked on the remote compilation platform. The build includes all units in the execution closure of Host_Units that have been coded since their last build.

   You can specify the name of an indirect file containing the complete list of units. For example, you can specify as an indirect file the build-list file that resulted from building a previous batch script (see step 5). The RCI then generates the batch script from this list of units.

   Do not specify units for different targets or units from views that are associated with different remote machines.

3. If any unit in Host_Units (or its closure) contains a pragma Main, set the Link-_Main_Units parameter to True to include linker commands in the build script.

4. Leave the Transfer_To_Target parameter True if you want to download units to the remote compilation platform when you build the batch script. This downloads units whether or not they have changed since the last download.

   Set this parameter to False if you want to use a different way of downloading units (see "Setting Switches That Control Batch Operations," above, and "Downloading Host Units," below).

   This option has no effect if the Effort_Only parameter is set to False.

5. Specify the desired filenames for the following parameters:

   - Host_Script_File: Specify the name of the batch-script file to be created on the host. The default value, "<DEFAULT>", causes the RCI to create a file called Batch_Script in the current context. If you specify a filename that already exists, the RCI does not overwrite the file and the build fails.
   - Remote_Script_File: Specify the full pathname of the batch-script file when it is downloaded to the remote compilation platform. If the script cannot be downloaded to the specified file, the command generates warnings and builds the batch script on the host.
   - Build_List_File: Specify the name of the *build-list file*, a host file that contains a list of the host units selected for remote compilation during the current build. The RCI creates this file only if the build succeeds; if the RCI cannot create this file, the build fails. The default value, "<DEFAULT>", causes the RCI to create a file called Units_To_Build in the current context.

     The units are listed in the order in which they need to be compiled. The build-list file can be used as an indirect file for regenerating a build or for uploading associated files for a particular build.

6. Set the Execute_Script parameter to True if you want to download and execute the batch script automatically on the remote compilation platform. When the batch script executes, its output is directed to an R1000 window. The Rci.Build-_Script command fails if remote execution fails; however, even if execution fails, the units referenced in the batch script retain their build timestamp.

   Leave this parameter False if you want to download and execute the batch script as a separate operation (see "Executing a Batch Script on the Compilation Platform," below).

7. Set the Effort_Only parameter to True if you just want to find out which units would be included in a batch script. The names of these units appear in the command's output log and no other action is taken—that is, no script is generated,

the build operation is not recorded in the RCI state information, nothing is downloaded to the remote compilation platform, and no Build_List_File is created.

8. Set the Make_Units parameter to True if you want to promote the specified units (and their closure) to the coded state on the host before generating a batch script.

   Leave this parameter False if you want to be notified if any unit is not yet coded.

9. Leave the Minimal_Recompilation parameter True if you want the batch script to include only obsolete units—that is, units (either specified or in the execution closure) that have been coded since they were last included in a batch script.

   Set the Minimal_Recompilation to False if you want the batch script to include the entire execution closure of the specified units, regardless of whether they are obsolete.

10. Press [Promote] to initiate the build.

Building Batch Scripts for Tape Environments

Use the Rci.Build_Script_Via_Tape command if you need to transfer files and scripts to the remote compilation platform using tape. The following command copies the specified units to the specified tape and includes a *host move script*, which contains the commands necessary for writing the files to the appropriate directories on the remote compilation platform:

1. Enter the Rci.Build_Script_Via_Tape command and press [Complete]. The command looks like this:

```
Rci.Build_Script_Via_Tape
    (Host_Units             => "<IMAGE>",
     Link_Main_Units        => True,
     Host_Script_File       => "<DEFAULT>",
     Remote_Script_File     => ">> FULL REMOTE NAME <<",
     Build_List_File        => "<DEFAULT>",
     Format                 => "R1000",
     Volume                 => "",
     Label                  => "rci_build",
     Logical_Device         => "rci:",
     Effort_Only            => False,
     Minimal_Recompilation => True,
     Make_Units             => False,
     Response               => "<PROFILE>");
```

2. Complete the tape-specific parameters:
   - Format: Specify the type of tape to be written, either ANSI, R1000, or R1000_LONG. For details, see the Format parameter of the Archive.Save command in the Library Management (LM) book of the *Rational Environment Reference Manual.*
   - Volume: Specify the tape volume name.
   - Label: Specify the tape label name. For details, see the Label parameter of the Archive.Save command in the Library Management (LM) book of the *Rational Environment Reference Manual.*
   - Logical_Device: Specify the logical device name for the tape device on the remote machine. The host move script, included on the tape, references the logical device rather than the physical device name.
3. Complete the remaining parameters as described in the previous section for the Rci.Build_Script command.
4. Press [Promote] to execute the command.

Checking the Build State

The Rci.Show_Build_State command displays a report that shows the compilation timestamp and the build timestamp for the specified units. By setting the Obsolete_Units_Only parameter to True, you can use this command to list just the obsolete units—that is, the units whose compilation timestamp is more recent than their build timestamp.

1. Enter the Rci.Show_Build_State command and press [Complete]. The command looks like this:

```
Rci.Show_Build_State
    (Host_Units          => "<CURSOR>",
     Execution_Closure   => False,
     Obsolete_Units_Only => False,
     Response            => "<PROFILE>");
```

2. Specify the Host_Units parameter with the names of the units for which you want build state information. Such information is shown only for coded units.
3. Set the Execution_Closure parameter to True if you also want to display information for the coded units in the execution closure of Host_Units.
4. Set the Obsolete_Units_Only parameter to True if you want information about obsolete units.
5. Press [Promote]. Output similar to the following appears:

```
unit              : !Proj.Sub.Rev1_Working.Units.A'Body
last coding time  : 92/04/06 09:47:31
last build  time  : 92/04/06 08:47:31
```

Downloading Host Units

If the host and remote compilation platform communicate using FTP, you can download units from the host as part of the Rci.Build_Script command (by setting the Transfer_To_Target parameter to True). Alternatively, you can download units as a separate step by entering the Rci.Transfer_Units command:

```
Rci.Transfer_Units (Units          => "<CURSOR>",
                    Remote_Machine => "",
                    Effort_Only    => False,
                    Response       => "<PROFILE>");
```

This command is especially useful during multimachine development, where a host view maps onto many remote libraries. In this scenario, you can generate the batch script and the build-list file once and then use the Rci.Transfer_Units command to download the units to the various remote compilation platforms. You can then download and execute the batch script on those platforms (see the next section).

Executing a Batch Script on the Compilation Platform

If the host and remote compilation platform communicate using FTP, you can initiate execution of the batch script from the host as part of the Rci.Build_Script command (by setting the Execute_Script parameter to True). Alternatively, you can

download and execute an existing batch script as a separate step by entering the Rci.Execute_Script command:

```
Rci.Execute_Script
   (Host_Script_File         => "<IMAGE>,
    Remote_Script_file       => ">> FULL REMOTE NAME <<",
    Remote_Machine           => "<DEFAULT>"
    Remote_Username          => "<DEFAULT>",
    Remote_Password          => "<DEFAULT>",
    Remote_Directory         => "",
    Effort_Only              => False,
    Display_Remote_Commands  => False,
    The_Key                  => "<DEFAULT>",
    Response                 => "<PROFILE>");
```

This command is especially useful during multimachine development, where you want to execute the batch script on multiple compilation platforms.

Retrieving Associated Files

In interactive mode, the RCI automatically uploads compilation results, such as object files and listings, when any unit is compiled. In batch mode, you must execute the following command as a separate operation to upload associated files (note that the host and remote compilation platform must communicate using FTP):

```
Rci.Upload_Associated_Files
    (Units       => "<CURSOR>",
     Effort_Only => False,
     Response    => "<PROFILE>");
```

Specify the Units parameter with a naming expression that describes the host units for which to upload compilation results. You can specify an indirect file that contains the names of the host units. For example, you can upload just the associated files of units selected for the last build by specifying the build-list file generated by the Rci-.Build_Script command.

This command may generate errors if a batch script was generated but never executed on the target.

TROUBLESHOOTING BATCH-MODE OPERATIONS

Batch compilations may fail on the target even when all the host units have coded successfully. This could happen for any of the following reasons:

- Units were not semantically correct for the target, even if they were installed on the host. (Semantic checking on the host cannot guarantee that units will compile on the target.)
- The target compiler has bugs and rejects valid Ada code.
- Not all supplier views/units are compiled on the target.

If a batch-script compilation fails, you can recover as follows:

1. Examine the output of the batch script to determine the problem. Determine which units failed to compile and why.

2. Fix the failed units on the host. This may mean demoting and promoting more than one unit.
3. Use the Rci.Build_Script command to generate a batch script for the units you fixed. Run this incremental batch script on the target.

   Note that some target compilers fail for all the units in a library if any unit in that library fails to compile. In such cases, generating an incremental script is not enough. To recover, you need to regenerate the batch script to include all the units in the view.
4. Enter the Rci.Build_Script with the following parameter values:
   - Host_Units: Specify the build-list file from the failed build as an indirect file—for example, `"_units_to_build"`. This causes the new script to contain all of the units that failed to compile, not just the ones you fixed.
   - Batch_Script: Specify a new filename.
   - Build_List_File: Specify a new filename.
   - Minimal_Recompilation: Set to False to force the build for all the specified units, even if they are still coded on the host.

   For example:

```
Rci.Build_Script
 (Host_Units             => "_units_to_build",    -- Failed build
                                                  -- units
  Link_Main_Units        => True,
  Transfer_To_Target     => True,
  Host_Script_File       => "batch_script_new",       -- New file
  Remote_Script_File     => ">> FULL REMOTE NAME <<",
  Build_List_File        => "units_to_build_new",     -- New file
  Execute_Script         => False,
  Effort_Only            => False,
  Minimal_Recompilation  => False,                 -- Force build
  Make_Units             => False,
  Response               => "<PROFILE>");
```

5 Maintaining File Consistency

Maintaining file consistency involves the following major and minor categories, which are addressed in this chapter:

- Consistency between views on the host
  - Copying and joining units from an RCI to an R1000 view
  - Copying and joining units from an R1000 to an RCI view
- Consistency between host and remote units
  - Keeping code consistent
  - Maintaining consistency in batch mode
  - Determining consistency of host and remote units
  - Replacing host units with updated remote units
  - Uploading a new remote unit to the host

CONSISTENCY BETWEEN VIEWS ON THE HOST

Consistency between files joined from an RCI view to an R1000 view is maintained automatically by CMVC as described in the Project Management (PM) book of the *Rational Environment Reference Manual.* Only one of a set of joined files can be checked out (Cmvc.Check_Out command) and edited at a time; checking out another of the set automatically performs updates on that file so that it matches the updated joined file.

Files that exist in a view used to create a new joined view (Cmvc.Make_Path or Rci_Cmvc.Make_Path) are automatically copied when the new view is created; those that are controlled by CMVC (Cmvc.Make_Controlled command) are joined with the copies in the new view.

After the views have been created and joined, however, it is often necessary to create new units in one view or the other; in this case, you must copy and join the units, as described below.

Copying and Joining Units from an RCI to an R1000 View

There are two ways to copy and join new units from an RCI to an R1000 view.

Updating All Units in a View

There are the Cmvc.Accept_Changes command, specifying Source and Destination views as described in Chapter 10, "Package Cmvc." All new units in the RCI view that

are controlled and checked in are copied into the R1000 view and joined with the file they were copied from. All existing joined units that are checked in are updated in the R1000 view as well.

Updating a Single Unit

Normally, you will use the Cmvc.Accept_Changes method described above. However, if you need to join a single unit without forcing updates in all joined units:

1. From any context, enter the Library.Copy command, specifying the following parameters:
   - From: Specify the pathname of the unit to be copied from.
   - To: Specify the pathname of the unit to be copied to.

   For example, the following command transfers a single object from an RCI view to an R1000 view:

```
Library.Copy
   (From => "!Users.My_File.Tr_Test.Working_Units.Foo",
    To   => "!Users.My_File.R1000_Directory.Working_Units.Foo");
```

   For a complete discussion of the Library.Copy procedure, see the Library Management (LM) book of the *Rational Environment Reference Manual.*
2. Use Cmvc.Join on the new R1000 unit and specify the old RCI view for the To_Which_View parameter.

Copying and Joining Units from an R1000 to an RCI View

Use Cmvc.Accept_Changes as you normally would to make two joined views consistent. In this case, when the units are copied into the RCI view and promoted to the coded state, they are processed as described in "What Happens During the Coding Step" on page 60 in Chapter 3, which includes downloading the units to the remote compilation platform.

CONSISTENCY BETWEEN HOST AND REMOTE UNITS

Using the integrated development cycle assumes that you do code development on the host environment, not the remote compilation platform. The RCI does not prohibit you from making changes to the source code of units located on the compilation platform that are associated with units on the host. Because of this, inconsistencies can arise between the host and remote units. This section addresses the management of issues involved with this process.

Keeping Code Consistent

The RCI keeps host and remote units consistent in the following manner:

- When you ask the RCI to promote a unit to the coded state, its host edit time is compared to its latest download time. If the edit time is later than the download

time, the unit's source code is downloaded automatically to the compilation platform. This ensures that the source code for a specific module is consistent.

- If a remote compilation is successful while promoting a unit to the coded state, the remote edit time of the newly downloaded unit is retrieved and saved in the RCI state file as the unit's download time.
- When you request a consistency check, the RCI determines how to compare in one of two ways, controlled by the Compare_Objects parameter:
  - If True, the remote unit is uploaded and compared to the host unit.
  - If False, the download time of the host unit is compared to the current edit time of the remote file currently associated with the unit.

Since code development takes place on the host, these consistency issues may require action:

- When a unit is promoted to the coded state, its remote source file is overwritten without performing the above consistency checking. "Determining Consistency of Host and Remote Units," below, describes how to check whether the remote source has changed.
- You must explicitly upload any new or changed units developed on the compilation platform, as described in "Replacing Host Units with Updated Remote Units" and "Uploading a New Remote Unit to the Host," below.
- The RCI checks whether the host unit has been updated since it was last downloaded only when the unit is promoted to the coded state. Therefore, changes made to the host file since the last download may not be compatible with changes uploaded from the remote file. Make sure that host and remote versions of a unit are not edited at the same time.
- The RCI does not perform consistency checks between listing files or object modules (including the remote program library). A successful promotion to the coded state with appropriate switches set is the best way to ensure consistency between associated files and their remote counterparts.
- The RCI does not perform consistency checking between executable modules in a given closure. Once an Ada main unit has been remote-linked, recompiling a unit in its closure does not affect the executable module. Be sure to always use Rci.Link to relink the Ada main unit after recompiling units in its closure.
- If the remote name associated with a unit is changed with the Rci.Set_Remote-_Unit_Name command, or if the remote filenames are changed, the RCI is no longer aware of the existing files on the remote compilation platform.

Maintaining Consistency in Batch Mode

Since there is no synchronous way on the host to gather information on the remote unit state when you execute a batch compilation script, it is more difficult to maintain compiled state consistency between host and remote units and libraries. You are responsible for ensuring that all units have been transferred and the batch compilation script has been executed on the remote compilation platform.

In particular, inconsistencies may develop for the following reasons:

- The script can be generated but never, or only partially, executed.
- All units are not transferred properly.
- Some units fail to compile remotely.

- Units are modified and compiled manually on the remote server.
- Regenerating a previous build can introduce new inconsistencies.

Using the RCI, you can regenerate a build in only one of two ways:

- Force recompilation by using Rci.Build_Script
- Regenerate the last build by rerunning the existing batch script

Determining Consistency of Host and Remote Units

The RCI provides operations to check whether remote text files have changed since they were last downloaded and to retrieve these units. These operations do not detect whether the host unit has changed since it was last transferred.

Note: *These operations check only source code; they do not detect changes to any listing, object, or executable files, nor do they check for obsolescence in the remote program library.*

To determine whether one or more units is consistent between the host and the remote compilation platform:

1. Enter the Rci.Check_Consistency command in a command window and press [Complete]:

```
Rci.Check_Consistency (Unit            => "<CURSOR>";
                       Compare_Objects => False,
                       Response        => "<PROFILE>");
```

2. Fill in the parameters as follows:
   - Unit: Specify a naming expression for the Ada units on the host to be compared to their associated units on the compilation platform. The remote name is determined as described in "Remote Files and Names" on page 57 in Chapter 3.
   - Compare_Objects: Specify a value. If False, the RCI compares the host-unit download time to the remote-unit edit time. If True, upload the remote unit and compare it to the host unit.

   Note: *This parameter can be set only to False for a DTIA customization. In a Telnet/FTP customization, Compare_Objects is ignored and is always set to True.*
3. Press [Promote]. The RCI compares the most recent download times of the specified units with the current edit time of the remote files, or it uploads the remote unit and compares it to the host unit. It displays a list of inconsistent units and appropriate messages for units that have never been coded or that have been deleted on the remote machine. If Compare_Objects is True, the RCI also displays the differences between the host and target files if they are inconsistent.

Since changes can be made to source code on both the host and the remote compilation platform, you must decide which file is the most recent and should replace the less recent file.

Replacing Host Units with Updated Remote Units

If one or more remote Ada units or non-Ada files should replace the existing host equivalents:

1. Use the Cmvc.Show command to ensure that you have checked out the units located on the host. Enter the Cmvc.Show command in a command window and press **[Complete]**:

```
Cmvc.Show (Objects  => "<CURSOR>",
           Response => "<PROFILE>");
```

Fill an @ into the Objects parameter to list all units in the current and subordinate libraries and press **[Promote]**. This generates output that includes the following information for all controlled objects:

Object Name	Generation	Where	Chkd Out	By Whom
UNITS.FIRST_UNIT	2 of 3	PROD_REV1_WORKING	Yes	TV
UNITS.SECOND_UNIT	2 of 3	PROD_REV1_WORKING	Yes	TV

If you have checked out the units you want to update, continue with step 3.

2. If someone else has checked out the units, the indicated user must check them in with the Cmvc.Check_In command before you can proceed. Then check out any units that are not checked out and need to be updated. Enter the following command in a command window:

```
Cmvc.Check_Out (What_Object => "");
```

Between the quotes, enter a naming expression for the units that are to be checked out. Press **[Promote]**. The CMVC command displays messages indicating the success of the operation.

For additional information on CMVC commands, see the Project Management book (PM) of the *Rational Environment Reference Manual.*

3. Enter the Rci.Accept_Remote_Changes command in a command window and press **[Complete]**. The result is:

```
Rci.Accept_Remote_Changes
           (Unit                => "<CURSOR>",
            Allow_Demotion      => False,
            Compare_Objects     => False,
            Remake_Demoted_Units => True,
            Goal                => Compilation.Coded,
            Response            => "<PROFILE>");
```

4. Fill in the parameters:

- Unit: Specify a naming expression indicating which host units and/or text files are to be replaced if necessary by the equivalent remote unit(s). If you want to update a secondary text file, specify the name of its associated primary.
- Compare_Objects: Specify a value. If True, consistency is checked by comparing host and remote objects; if False, host and remote timestamps are checked.

If a text file is specified for Unit, then the Allow_Demotion, Remake_Demoted-_Units, and Goal parameters are ignored.

If an Ada unit is being updated, fill in the following parameters:

- Allow_Demotion: Specify a value. To be updated on the host, an Ada unit must be in the source state. If this is False, the command does not demote host units. If demotion is required, the command displays a message and no changes are accepted for that unit.
- Remake_Demoted_Units: Specify a value. If True, any host units demoted to make changes are repromoted to the state noted below.

- Goal: Fill in a valid Compilation.Unit_State value (such as Compilation.Installed); the RCI attempts to repromote units that are demoted during the update process to the indicated state. Units can be promoted only to the installed state; requesting Compilation.Coded gives a warning to this effect.

5. Press [Promote]. The specified units are updated and promoted to the indicated state only if the associated remote units have changed since they were last downloaded; otherwise, the host units remain untouched. Units on the remote compilation platform remain unaffected.

Uploading a New Remote Unit to the Host

If you have created new Ada units on the remote compilation platform that should be maintained in the RCI view, you can upload them either one at a time or as a group. If you want to upload a single unit, use the Rci.Upload_Unit command. If you have several units, use the Rci.Upload_Units command; this allows you to define remote and host unit names in a text file which is selected by the Upload_Specification_File parameter.

To upload a single new Ada unit from the remote machine:

1. Enter the Rci.Upload_Unit command and press [Complete]:

```
Rci.Upload_Unit
           (Remote_Unit_Name    => ">>REMOTE UNIT NAME<<";
            Into_View           => "<CURSOR>";
            Upload_To_Text_File => False;
            Host_Text_File_Name => "";
            Response            => "<PROFILE>");
```

2. Fill in the parameters as follows:
   - Remote_Unit_Name: Specify the simple name of the Ada unit on the remote machine.
   - Into_View: Specify the view or directory within the view where the specified remote unit exists.
   - Upload_To_Text_File: See Chapter 7, "Using Non-Ada Units," for further information.
   - Host_Text_File_Name: Leave null.
3. Press [Promote]. The RCI uploads the remote unit onto the host into an Ada unit in the source state, which is named according to the standard Rational Environment Ada unit names. If it cannot be parsed, it remains a text file with a name such as Temp_90_12_11, where the final digits represent the date the file is created.
4. Use the Cmvc.Make_Controlled command to control the new unit.

To upload more than one new unit from the remote machine to the host (using an Upload_Specification_File):

1. Create a text file on the host by pressing [Create.Text] in the view that will contain the uploaded units. This file will contain information about the name the RCI uses to find a unit on the remote machine and the new name on the host where it will store the uploaded unit.

   In your open text file, create one text line for each file you want to upload. Enter one or two name fields on each line of text:
   - Remote unit name: Specify the name of the unit on the remote compilation platform.

- Host text filename: Specify the new host name for the uploaded unit. If this field is left blank, the command attempts to upload the remote unit into an Ada name on the host. The RCI uses the naming scheme for remote units that has been defined by your extension to create the Ada unit name on the host.

Separate the remote unit name from the optional host text filename by one or more spaces. A sample file appears as follows:

```
x.c             x_c
new_unit.c      unit_in_c
y_s.ada
```

Using this sample file, remote file x.c is copied into host text file X_C; remote unit new_unit is copied into host text file Unit_In_C; remote file y_s.ada is copied into host Ada unit Y'Spec.

If a line of the specification file does not provide a host text filename, the RCI attempts to upload the file into an Ada unit.

2. Enter the Rci.Upload_Units command and press [Complete]:

```
Rci.Upload_Units
           (Upload_Specification_File => "";
            Into_View                 => "<CURSOR>";
            Response                  => "<PROFILE>");
```

3. Fill in the parameters as follows:
   - Upload_Specification_File: Specify the host text file, created in step 1, which contains the list of units to upload.
   - Into_View: Specify the view in whose associated remote directory the specified remote units exist.
4. Press [Promote]. The RCI uploads the remote units listed in the specification file onto the host. Use the Cmvc.Make_Controlled command to control the new units.

6 Library Management

This chapter describes activities involved with managing host and remote libraries. Many of the approaches in this chapter depend on the specific characteristics of your customization extension. Your extension may or may not handle libraries in the manner described in the examples. Use the examples in this chapter as general guidelines and apply them to your specific extension. Refer to the user's guide for your extension or to guidelines provided by your site customizer for additional information.

This chapter includes descriptions of the following:

- RCI library model
  - Overview
  - Definitions
  - Examples
  - Limitations and restrictions
- Management of remote libraries
  - Automatic creation
  - Explicit creation
  - Example of library creation
  - Removing remote libraries
- Imports
  - Adding imports
  - Removing imports
  - Keeping imports consistent
  - Imports example
- RCI state information
- Management of subsystems and views
  - Removing RCI views
  - Creating releases of views

RCI LIBRARY MODEL

The RCI library model is similar to the native R1000 library model. It is limited, however, by the restrictions of the remote compilation platform's library model.

Overview

The RCI maps RCI combined views to libraries under the remote operating system. RCI library-management operations (Make_Path, Import, and Remove_Import, for example) can be extended optionally to perform remote library-system operations so that you do not have to manage remote libraries as a separate task from managing host libraries.

Note: *If you choose to run without, or if the extension of the RCI does not implement, remote library management, you will be responsible for the management of the state of any remote libraries.*

To enable these extensions, your system administrator and customizer must modify the extension. See the "Enabling Remote Extensions Management" in Chapter 2.

The RCI normally is used only with combined views. Combined views generally map to remote operating-system libraries in a straightforward manner. The Rational Environment has the notion of a current working library, as do most library-management systems. Also, the Rational Environment has a way of defining static imports (CMVC maintains a list of imports for each combined view on the host), and many library-management systems maintain an equivalent list of sublibraries to use when resolving external references on the remote compilation platform.

Definitions

When this manual refers to a *remote library*, it includes a directory under the remote operating system, a presumed associated or enclosed target-compiler program library, and, for some extensions, a remote import list. The RCI uses these parts of the remote library as follows:

- Ada source is maintained in (or downloaded to) the *remote directory*.
- Ada source is compiled into the *program library;* that is, the program library contains a compiled representation of units.
- The *remote import list*, which is maintained in the library list file, specifies the sublibraries (imports) to be used during compilation for some extensions. (Other approaches to imports include unit-by-unit definitions or hierarchial directory structures.)

A Rational view is analogous to the remote library: the Units directory of the view contains all of the source, and information about compiled units is an implicit or hidden part of the view.

The implementation and names of these objects are discussed in "Automatic Creation" on page 89.

Examples

The following examples illustrate possible library-structure mappings between host subsystems and remote libraries.

Figure 6-1 shows multiple views, each in a different host subsystem, mapped to a single remote library. Views from several subsystems in the same closure can map to a remote library in a many-to-one relationship.

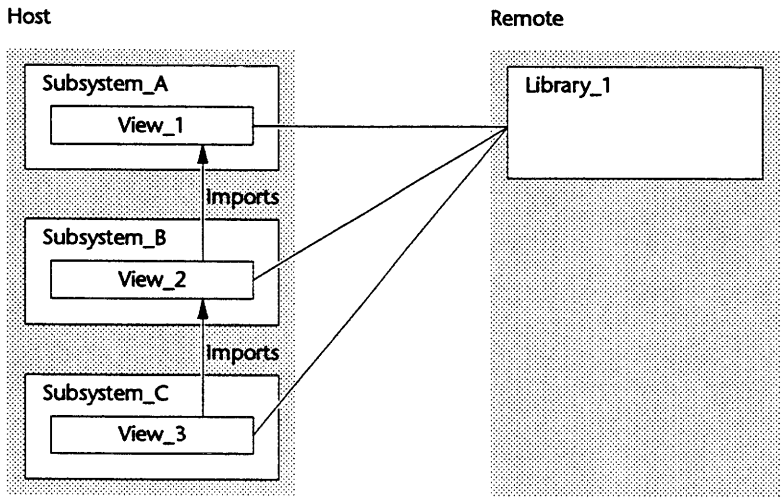

Figure 6-1 Multiple Subsystems with Single Remote Library

Note: *For some extensions, the closure of views must all be mapped to the same remote library, and no other remote libraries can be imported into this library. If the closure is not restricted to this single library, the remote-library list file will not accurately reflect remote imports, and compilation errors will result.*

Figure 6-2 shows a one-to-one mapping between host views, remote libraries, and imports.

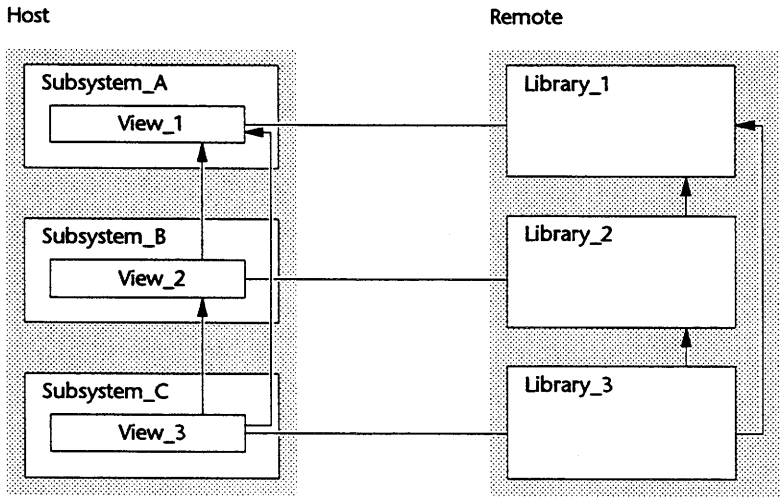

Figure 6-2 Multiple Subsystems with Multiple Remote Libraries

Limitations and Restrictions

The RCI library model essentially supports a one-to-one mapping of host views to remote libraries. The following restrictions apply to mapping host views to remote library structures:

- All RCI development must be done in combined views; the spec/load-view model is not supported.

- More than one view in the same host subsystem cannot be mapped to a single remote library; an attempt to do this results in an error. Figure 6-3 shows this erroneous mapping.

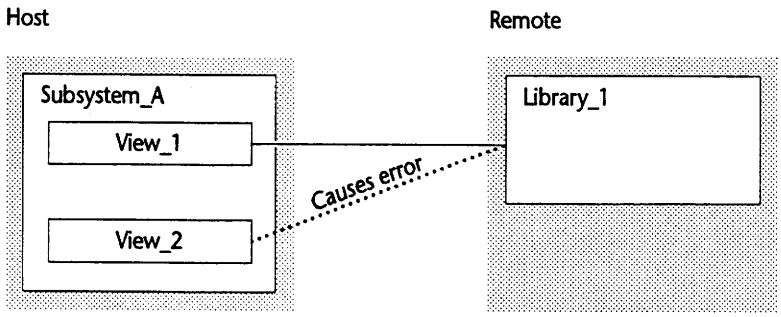

Figure 6-3 Multiple Views in a Single Subsystem

- Views from several subsystems where the views are part of the same closure can be mapped to a single remote library. However, mapping several of these views to one remote library and others to another remote library can result in inconsistencies of imports. Figure 6-4 shows this undesirable mapping.
- A host view cannot map to more than one remote library because the remote library for a view is determined as described in "Remote Directory" on page 22 in Chapter 2. Figure 6-5 shows this impossibility.
- Imports in remote program libraries must be expressed in a manner that is consistent with imports in host views; this should occur automatically if import information is controlled from the host using CMVC and RCI commands.

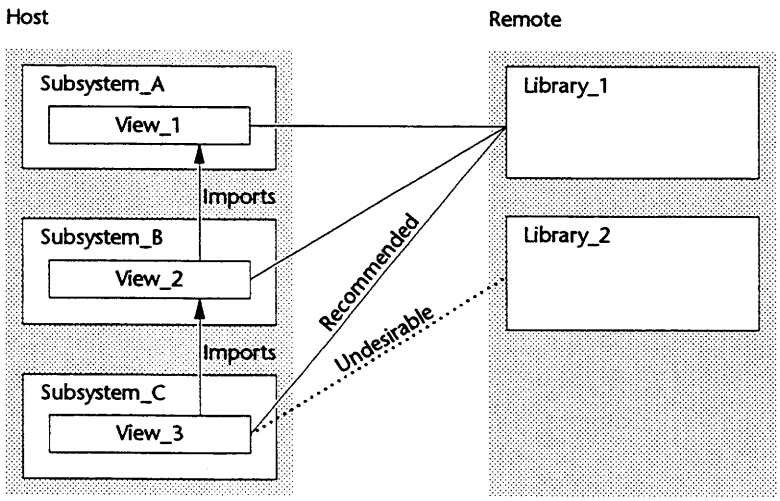

Figure 6-4 Views in the Same Closure

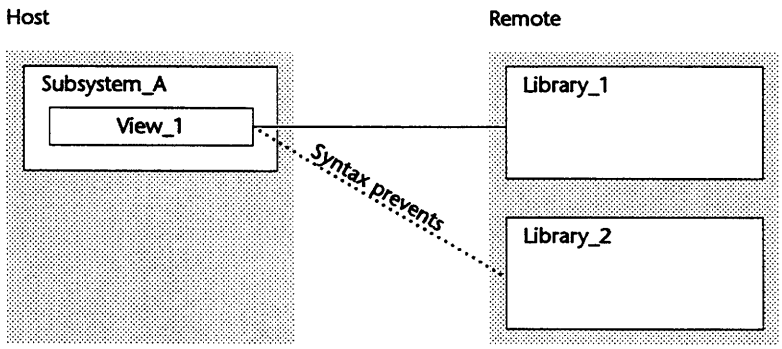

Figure 6-5 Multiple Remote Libraries

MANAGEMENT OF REMOTE LIBRARIES

If your extension has enabled RCI automatic library management, you have set the Session_Rci.Auto_Create_Remote_Directory switch to True, and you have set up the information required for remote communication as described in "Setting Up Remote Communications" on page 19 in Chapter 2, the RCI creates a remote library whenever you create an RCI view. For more information, see the user's guide for your extension. Depending on your extension, the RCI creates the nested directories on the remote platform as well as the specified remote directory. If your extension does not support this operation, you must create the libraries yourself.

In interactive mode, the RCI cannot promote units in an RCI view to the coded state unless it can download the units to and compile them on the remote compilation platform.

Automatic Creation

The CMVC view-creation commands (Initial, Make_Path, Make_Subpath) and their corresponding Rci_Cmvc commands automatically build the remote library under the conditions described in "Creating a Subsystem and an RCI View" on page 37 in Chapter 2.

If the remote library already exists, it is used as is.

Ifthe remote library does not exist, the RCI creates it as a directory whose full pathname is the value created by the default switch-naming scheme for the Cmvc command or the Remote_Directory parameter of the Rci_Cmvc command used to create the view. Within that directory, the RCI also creates the equivalent of these items as determined by the customization:

- Remote program library
- Remote import list

Imports in the remote import list are derived from the CMVC imports that exist at the time of the library's creation. If no CMVC imports exist, the remote import list is created with the minimal entries as described in the next subsection. This list is modified whenever CMVC imports are modified using CMVC or RCI commands.

See "Imports" on page 94 and Appendix C for further discussion.

Explicit Creation

If the RCI does not create a remote library when you create an RCI view—for example, because the remote username and password were incorrect, or because the remote library-management job was not running—you must create a remote directory and the appropriate program library and, if necessary for the extension, a remote import list. Use the Rci.Build_Remote_Library command or create the library manually using the library-creation procedure for your extension.

Building a Remote Library

The Rci.Build_Remote_Library command builds the remote library for an existing RCI view. The remote library consists of both the remote directory, where source is stored, and the program library into which it is compiled. If appropriate, this command also creates the remote import list.

Use the following steps to explicitly create a remote library:

1. Enter the Rci.Build_Remote_Library command in a command window and press [Complete]:

```
Rci.Build_Remote_Library
          (View             : String := "<CURSOR>";
           Remote_Machine   : String := "<DEFAULT>";
           Remote_Directory : String := "<DEFAULT>";
           Response         : String := "<PROFILE>");
```

2. Fill in the parameters as follows:
   - View: Specify the host view for the remote directory.
   - Remote_Machine, Remote_Directory: Specify these values. If you use the defaults, the RCI takes the values from the view's compiler switches. If these values are not in the view's compiler switches, the view does not have an existing remote library, and you must provide them here.
3. Press [Promote]. This connects to the remote machine and creates the remote program library, using the specified remote machine and remote directory.

This process works only if the Make_Path_Postprocess RCI customizable library extension has been implemented. This library extension may or may not be implemented from one RCI extension to another.

For more information on Build_Remote_Library, see Chapter 8.

Rebuilding an Existing Remote Library

If an existing remote library becomes corrupted, the Rci.Rebuild_Remote_Library command can rebuild it.

Use the following steps to rebuild an existing remote library:

1. Enter the Rci.Rebuild_Remote_Library command in a command window and press [Complete]:

```
Rci.Rebuild_Remote_Library
    (View                 : String  := "<CURSOR>";
     Remake_Demoted_Units : Boolean := True;
     Remote_Machine       : String  := "<DEFAULT>";
     Remote_Directory     : String  := "<DEFAULT>";
     Response             : String  := "<PROFILE>");
```

2. Fill in the parameters as follows:
   - View: Specify the host view for the remote directory.
   - Remake_Demoted_Units: Specify the value. If True, all units and dependents are recoded.
   - Remote_Machine, Remote_Directory: Specify these values. If you use the defaults, the RCI takes these values from the view's compiler switches. If these values are not in the view's compiler switches, you must provide them here.
3. Press [Promote]. The process destroys the existing remote library specified by the View parameter, demoting all units in that view and their dependents. Then it rebuilds the remote library.

Creating a Remote Library

Use the procedure defined for your remote operating system to manually create a remote library on the remote machine. Refer to the user's guide for your remote operating system and the user's guide for your extension for more information. Your RCI customizer provides a list of actual steps to perform remote library management. Space for this information is provided in Appendix C.

Note: *The following steps and commands are specific to a sample customization for the RS6000 target. The actual steps and commands required vary by operating system.*

The following example describes the steps to create a remote library:

1. Log into your remote machine.
2. Enter:

```
$ mkdir rci_test
$ cd /u/YOURNAME/rci_test
```

   The mkdir command builds a new directory with the name rci_test in the context in which you logged in (here it is assumed that your home directory is /u/YOURNAME); the cd command changes your context to this new directory.
3. Create the remote import list by entering:

```
$ cat > alib.list
```

   This presents a blank line into which you must enter the following two lines:

```
working
/usr/lpp/ada/lib/libada
```

   Press [Control][D] to terminate the input to the cat command and complete the creation of the remote import list, which contains a list of libraries to be used by the target compiler when resolving compilation dependencies. You have just placed two entries into this file:

- working: the local program library
- /usr/lpp/ada/lib/libada: the target-compiler LRM-predefined units

4. Create the local program library by entering:

```
$alibinit
```

5. Log off the remote machine by entering:

```
$logout
```

For more information on target program-library management, see the operating-system user's guide for your extension.

Example of Library Creation

This example works as described if the Rci.Host_Only switch is set to False and the operation mode for the view is Interactive.

Assume that there is currently an R1000 subsystem, Dog_And_Pony_Show, which contains a view called R1000_Devel_Working. This view's Units directory contains a specification and a body for an Ada unit named Dogs. On the remote compilation platform, which is listed in the transport name map as Tilden, there exists a directory called /u/jones, which is the home directory when you log into the remote machine with the remote username Fred and the remote password Farsides. This setup is shown in Figure 6-6.

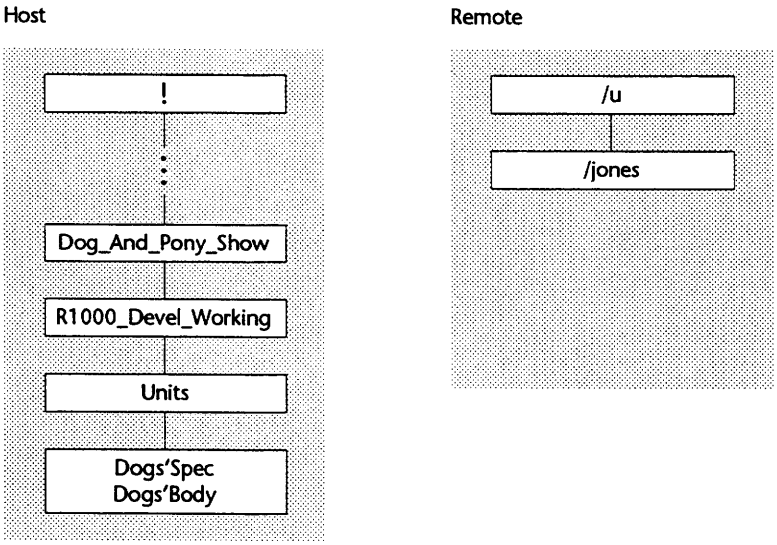

Figure 6-6 Initial Library Setup

The following steps show how the RCI creates a remote library for the RS6000 when a host view is created:

1. Set your Session_Ftp.Username switch to `fred` and your Session_Ftp.Password switch to `farsides`. (Or use the Remote_Passwords file as described in "Specifying Remote Login Information" on page 20 in Chapter 2.)
2. Position the cursor on the existing viewname.
3. If the default switch-naming scheme for remote directory and machine are correctly set, enter the Cmvc.Make_Path command. If you need to specify the remote

machine or directory, enter Rci_Cmvc.Make_Path. See "Creating New RCI Views" on page 39 for a full description of parameters for these commands.

Enter one of the following commands (with selected parameters shown):

```
Cmvc.Make_Path
         (New_Path_Name => "Target_Devel",
          Model         => "!Model.Custom_Key",
          Goal          => Compilation.Coded);

Rci_Cmvc.Make_Path
         (New_Path_Name    => "Target_Devel",
          Remote_Machine   => "Tilden",
          Remote_Directory => "/u/jones/dog_and_pony-_show",
          Model            => "!Model.Custom_Key",
          Goal             => Compilation.Coded);
```

4. Press [Promote].

   This action does the following:

   - Creates a new host view named Target_Devel_Working and the remote library structure
   - Copies the units from the original view into the new view
   - Downloads each unit from the new view to the remote machine
   - Invokes the remote compiler on each unit as it is downloaded
   - Uploads the resulting object modules into the .<Obj> files on the host (if the extension uploads object associated files)
   - Promotes the host units to the coded state

The result looks as shown in Figure 6-7; actual suffixes vary for your extension.

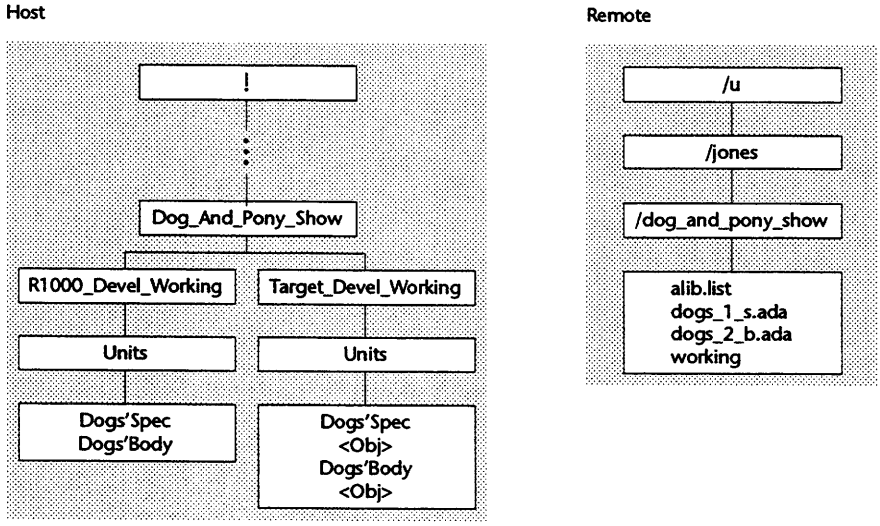

Figure 6-7 Libraries After Make_Path

Removing Remote Libraries

The Rci.Destroy_Remote_Library command allows you to remove the remote library associated with a host view.

Use the following steps to remove a remote library:

1. Enter the Rci.Destroy_Remote_Library command in a command window and press [Complete]:

```
Destroy_Remote_Library
              (View     : String := "<CURSOR>",
               Response : String := "<PROFILE>");
```

2. Fill in the View parameter to indicate the view whose remote library will be destroyed.
3. Press [Promote]. This destroys the remote program library, Ada source, and working directory associated with the specified view and demotes host units.

This process works only if the Destroy_View_Postprocess RCI customizable library extension has been implemented. This library extension may or may not be implemented from one RCI extension to another.

See Chapter 8 for information about the Rci.Destroy_Remote_Library command.

IMPORTS

The Rational CMVC library model has the notion of an import list, which specifies all of the subsystems (spec views or combined views) to make visible inside the importing view. Each extension has a different method for handling imports. Under some target-compiler models (for example, an RS6000), each remote library also has an *import list*, often implemented as a library-list text file, whose default name is determined by your extension. This is called the *remote import list*.

The required content of the remote import list varies by the remote operating system and Ada compiler. Generally, the remote import list contains three classes of information:

- The name of the current remote program library
- External (sublibrary) names that effectively specify imports
- The home of the target-predefined library

For example, as shown in "Explicit Creation" on page 90, this might contain:

```
working
/usr/lpp/ada/lib/libada
```

The RCI automatically maintains remote imports if your extension has enabled remote library management. Adding imports to a host view using RCI commands can cause imports (sublibraries) to be automatically added to the remote library's import list. Removal of imports from a host view using RCI commands causes automatic removal of sublibraries from the remote import list.

Note: *RCI views can import only other RCI views with the same target key.*

Adding Imports

The Cmvc.Import command adds an import to both the host view's import list and the remote import list. Each time you call Cmvc.Import, it updates or rebuilds *(refreshes)* remote imports for the associated remote library.

To add an import to a view's import list, which involves updating the host imports and the remote import list:

1. Enter the Cmvc.Import command in a command window and press [Complete].
2. Fill in the parameters as described in the Project Management book (PM) of the *Rational Environment Reference Manual* for Cmvc.Import.
3. Press [Promote].

Imports are created and remote imports are refreshed.

See "Keeping Imports Consistent," below.

Removing Imports

To remove imports from both the host view and the remote-library import information (import list or pathname), depending on your extension, use the Cmvc.Remove_Import command. Parameters for the Cmvc.Remove_Import command are described in the Project Management book (PM) of the *Rational Environment Reference Manual.*

Different extensions handle imports in different ways. For example, with the Rs6000 extension, each time that you call Cmvc.Remove_Import, the remote import list for the associated remote library is completely refreshed. Check with your customizer for more information.

See "Keeping Imports Consistent," below.

Keeping Imports Consistent

Import information is stored in the CMVC import list and, depending on the extension, in the remote import list on the remote machine.

The remote import list is completely refreshed, providing that remote library management is enabled and a remote connection can be made, whenever any of these commands is used and when Session_Rci.Auto_Create_Remote_Directory is True and Operation_Mode is Interactive:

- Cmvc.Make_Path or Rci_Cmvc.Make_Path
- Cmvc.Make_Subpath or Rci_Cmvc.Make_Subpath
- Cmvc.Initial or Rci_Cmvc.Initial
- Cmvc.Import
- Cmvc.Remove_Import
- Cmvc_Release or Rci_Cmvc.Release

If the above commands cannot update the remote import list as well as the view's import list, the RCI issues a warning and only the view's import list is updated. In this case, you will need to correct the problem and explicitly refresh the remote import list with the Rci.Refresh_Remote_Imports command.

The consistency between CMVC imports and the remote import list can be verified with the Cmvc.Information or the Rci.Show_Units command.

Imports Example

Assume the same setup as described in "Example of Library Creation" on page 92. Assume the existence of another host subsystem, Main_Subsystem, which also contains an RCI view named Target_Devel_Working. The remote program library for this view is the default, working. This view maps onto the remote library /u/jones/main_subsystem/devel_working.

Use the command:

```
Cmvc.Import
 (View_To_Import => "!...Main_Subsystem.Target_Devel_Working",
  Into_View      => "!...Dog_And_Pony_Show.Target_Devel_Working");
```

This adds the View_To_Import to the import list of the view specified in Into_View. It also looks in the state file of the imported view to find its associated remote program library and adds that name (in this case, working) to the remote import list for Into_View.

RCI STATE INFORMATION

Information about the RCI state is stored in each view. Knowledge of the mechanisms for storing state information is not generally important for the user. However, you may find it useful to know:

- Which files are used by the RCI and should not be touched
- When the information is updated

Where State Information Is Stored

RCI state information is stored in these binary files in the Tool_State directory:

- Rci_State_Batch_Times
- Rci_State_Download_Times
- Rci_State_Target_Names
- Rci_State_Unit_Options

In addition, the remote library name, which is considered to be part of the state information, is maintained as the value of the Rci.Remote_Library switch.

The view-creation commands create the RCI state automatically when they create the host view; the files should never be deleted or altered by the user. If the files are damaged or destroyed, you can use the Rci.Refresh_View command to rebuild them.

Secondary information is not stored as part of the state information.

When State Information Is Updated

The host and remote names of Ada units and some additional information is *registered* in (added to) the state information for any new Ada unit when it is promoted to the installed state. In addition, its dependencies are calculated and registered.

Whenever any Ada unit is promoted to the coded state, the consistency information is updated.

Demoting a unit has no effect on the state information.

Note: *Because the state files are locked while being updated, simultaneous installing or coding steps (and the execution of some commands) may fail with a state-file lock error. If this happens, repeat the command.*

Unit-state information is updated as shown in Figure 6-8.

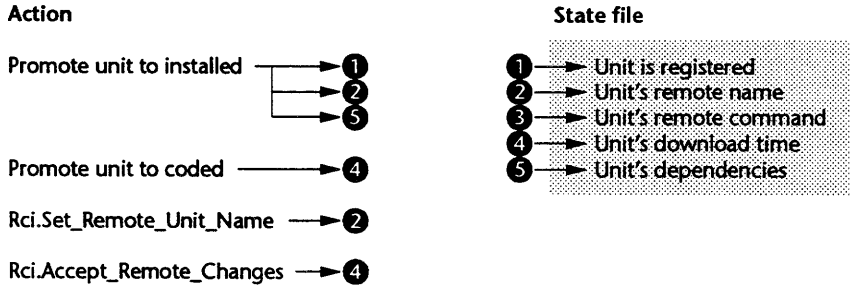

Figure 6-8 State-Information Updates of Unit-Level Information

MANAGEMENT OF SUBSYSTEMS AND VIEWS

Create and use host subsystems and views as described in the Project Management (PM) book of the *Rational Environment Reference Manual.*

The Session_Rci.Auto_Create_Remote_Directory switch controls whether or not the view-creation commands automatically build remote directories and libraries when you create views. See "Remote Directory" on page 22 for more information.

Package Rci_Cmvc provides corresponding view-management commands to CMVC for cases in which you cannot take advantage of the default switch-naming scheme to control remote-machine and remote-directory names.

Use package Rci commands to manage remote libraries.

Chapter 2, "RCI Setup Operations," provides an overview of subsystems, views, and remote libraries in "Setting Up Model Worlds" and "Preparing to Set Up Library Structures." Chapter 2 also discusses the creation and maintenance of RCI views and remote libraries in "Setting Up Library Structures."

Removing RCI Views

The Cmvc.Destroy_View command lets you destroy the host view or destroy the host and remote views at the same time depending on what value you have set for the Rci.Host_Only switch. If Rci.Host_Only is set to True, the command destroys the host view but leaves the remote library intact on the remote compilation platform. If False, it destroys the remote library and then the host view. For additional parameters, see the Cmvc.Release command.

The default command is:

```
Cmvc.Destroy_View (What_View                  => "<SELECTION>",
                   Remote_Clients             => False,
                   Destroy_Configuration_Also => False,
                   Comments                   => "",
                   Work_Order                 => "<DEFAULT>",
                   Response                   => "<PROFILE>");
```

Normally, when a view is destroyed, CMVC maintains a history of the view's existence, which prevents a new view from being created with that view's name. If you need to destroy and then recreate a view of the same name—for example, because you could not connect to the remote machine during the creation operation—set Destroy_Configuration_Also to True when destroying the view.

Creating Releases of Views

A released view is a copy of a view that is frozen in time. The Cmvc.Release command allows a simultaneous release to be made on the remote compilation platform. The corresponding command, Rci_Cmvc.Release, has additional parameters to specify remote machine and directory names.

The default command (with selected parameters) is:

```
Cmvc.Release (From_Working_View          => "<CURSOR>,
              Release_Name               => "<AUTO_GENERATE>",
              Level                      => 0,
              Views_To_Import            => "<INHERIT_IMPORTS>",
              Create_Configuration_Only  => False,
              Compile_The_View           => True,
              Goal                       => Compilation.Coded,
              Comments                   => "",
              Work_Order                 => "<DEFAULT>",
              Volume                     => 0,
              Response                   => "<PROFILE>");
```

General Release Strategy

Because units in a frozen view cannot be demoted, it is important that you release views imported by other views first. For example, if View_A imports View_B which imports View_C, View_C must be released first, View_B second, and View_A last.

As you make each release, you must change any existing imports to reference the appropriate released views.

Remote Releases

If a value exists for Remote_Directory (either in the library switch or in the parameter when using the Rci_Cmvc command) and a connection can be made to the remote machine, then a remote release is created as well as a host release. The creation of the remote release works exactly as creating a remote library does in Cmvc.Make_Path. This assumes that the Rci.Host_Only switch is set to False and the operation mode for the view is Interactive. The RCI takes the following action:

- Creates the new host view
- Creates the specified remote library
- Downloads all units (and secondaries) to the new remote library

- Compiles the units
- Uploads all appropriate files to the new view
- Freezes the new view

Note: *The RCI does not freeze the new remote library. You may want to use facilities in the remote operating system to change the access security on the remote library to allow read-only access so that it is effectively frozen.*

If Session_Rci.Auto_Create_Remote_Directory is False, no attempt is made to create a remote directory. Units in the released view will exist only in the installed state. All secondary referencers are moved to the new view and frozen. Other associated files are not moved to the new view.

This latter variety of release is useful when the purpose is to catch a snapshot of source state at an instant in time.

7

Using Non-Ada Code with the RCI

This chapter addresses how you can create and maintain non-Ada code, such as assembly-language source or C-language source, in the Rational Environment, process it from the Rational Environment, and keep it in a consistent state with equivalent code on the remote compilation platform, using the features of CMVC and the RCI. The operations perform as described in this chapter when the operation mode is Interactive.

This chapter includes:

- Using non-Ada units
- Creating a non-Ada unit
- Viewing and changing secondaries
- Processing secondaries

USING NON-ADA UNITS

The RCI provides a method to use and maintain non-Ada units, which are text files that contain source text for other assemblers, language compilers, or macro processors. The RCI associates these files with Ada units to control processing order on the remote compilation platform. Such a file is called a *secondary*; the associated Ada unit in the host environment is called a *primary*. The primary controls compilation dependencies on the remote compilation platform.

The RCI associates a primary with a non-Ada unit *(a secondary text file)*, which contains the non-Ada source text, through a *secondary referencer* file. This file contains information about the remote name, remote process command, and host name of a secondary text file, including the following:

- The name of the secondary text file associated with the primary
- The remote filename into which the secondary text file is downloaded
- The remote command to execute on the remote compilation platform after the secondary text file is downloaded
- The download time of the secondary text unit on the remote compilation platform

The name of the associated file (a *pointy file*) for the secondary referencer is formed as follows: Primary_Unit_Name.<Secondary_*Secondary_Text_File_Name*>.

The RCI controls the processing of the primary unit through a Boolean *Process_Primary* value. This value determines whether the primary unit is downloaded and

processed on the remote compilation platform. That information is stored in the *secondary state file.* The RCI creates a single secondary state file for each primary with associated secondaries. Each time you use Rci.Create_Secondary with a primary, the Process_Primary value is updated in the secondary state file. The name of the secondary state file has the form: Primary_Unit_Name.<Secondary_State>.

Information about the association between the primary and secondary is preserved in the secondary referencer and the secondary state file.

The example on page 106 shows the units in the host environment.

A primary can have many secondaries; a secondary can be associated with only one primary.

CREATING A NON-ADA UNIT

This section describes the steps necessary to create a non-Ada source file (a secondary unit) that is associated with an Ada unit (a primary).

Creating the Controlling Ada Unit (The Primary)

Since the Rational compilation system understands only Ada units as being compatible objects, you must create an Ada unit, either a spec or a body, to represent any non-Ada code on the host. This *primary* determines compilation ordering.

To create an Ada primary:

1. Move to the Units directory of the view in which the non-Ada code will reside.
2. Create an Ada unit in that view using the Common.Object.Insert command (press [Object] [I]). The Rational Environment creates an Ada unit and opens it in a window for editing.
3. Enter valid Ada code and any *with* clauses needed to determine the secondary's compilation dependencies. Good naming practices suggest that the name of the Ada unit be related to the name of the subprogram that will be contained in the secondary; for example, if the program in the secondary is named Foo, also call the Ada unit's main routine Foo:

   ```
   procedure Foo;
   ```

   A primary can be either an Ada spec or body; it cannot be an Ada separate subunit. Promoting the primary causes the secondary to be downloaded and processed on the remote machine if the operation mode is Interactive. The primary itself is downloaded only if the Process_Primary flag is set to True when the secondary is created, or by specifying so with the Rci.Set_Process_Primary command.
4. Press [Promote] to save the Ada unit in its installed state.

Creating the Non-Ada Source File

Enter non-Ada source code (the secondary text file) on the host.

If you have first created it on the remote compilation platform, see "Changing Text from the Remote Machine" on page 105.

Enter non-Ada source code into a Rational text file as described below. In addition, you must provide instructions that tell the remote operating system what to do with the code when its primary is promoted to the coded state.

1. Using the remote operating-system command syntax, write down the command to assemble or compile the code on the remote machine. This command will contain the name of the secondary. For example, to assemble the file named Foo, this might be:

   ```
   as foo
   ```

   or, to invoke the C compiler on the file:

   ```
   /u/compiler_directory/cc -c foo.c
   ```

2. Move to the Units directory of the RCI view containing the controlling Ada primary unit created as described in the previous subsection.
3. Position the cursor on the Ada primary.
4. In a command window, enter `Rci.Create_Secondary` and press **[Complete]**:

   ```
   Rci.Create_Secondary (Primary_Unit    => "<CURSOR>";
                         Command         => "";
                         Secondary_Text  => "";
                         Remote_Name     => "<DEFAULT>";
                         Process_Primary => False;
                         Response        => "<PROFILE>");
   ```

5. The default Ada unit (Primary_Unit) is the one on which the cursor is located. Alternatively, enter the Ada unit's name for Primary_Unit, as in this example:

   ```
   Rci.Create_Secondary (Primary_Unit    => "Foo'Spec;"
                         Command         => "";
                         Secondary_Text  => "";
                         Remote_Name     => "<DEFAULT>";
                         Process_Primary => False;
                         Response        => "<PROFILE>");
   ```

6. Enter the remote command that you noted in step 1. For example, to compile the file with the compile command shown in step 1:

   ```
   Rci.Create_Secondary (Primary_Unit    => "Foo'Spec";
                         Command         => "cc -c foo.c";
                         Secondary_Text  => "Foo_c";
                         Remote_Name     => "foo.c";
                         Process_Primary => False;
                         Response        => "<PROFILE>");
   ```

   When the primary is promoted to the coded state and the mode is Interactive, this causes the specified command to be issued on the remote machine.

7. Press **[Promote]**. The RCI takes the following actions:

   a. If a text file that has the same name selected for the secondary already exists, that file is used for the secondary. If the text file does not exist, the command creates it in the current view and uses it for the secondary.

   b. If the primary is in the coded state, it is demoted to the installed state.

   c. A secondary referencer file is created.

   d. A secondary state file is created if one does not already exist.

   e. The secondary text file is frozen, preventing editing except with the Rci.Edit_Secondary command.

f. A message is generated indicating the success of the action and specifying the name and remote name (see "Remote Files and Names" on page 57 in Chapter 3) that have been assigned to the secondary.

8. If you are maintaining your files under CMVC source control, use Cmvc.Make_Controlled to control the secondary file.
9. Edit the secondary using the instructions in "Changing the Secondary File and Commands," below.

VIEWING AND CHANGING SECONDARIES

Information about secondaries and their associated remote compiler commands is saved in the secondary referencer and secondary state files.

Viewing Secondary Relationships

The Rci.Show_Secondary command displays a set of one or more Ada primaries and information about the secondaries associated with each primary, their remote names, the status of their Process_Primary flags, and the remote operating-system command associated with them. For example, to display information for all primary units in the current view:

```
Rci.Show_Secondary (Primary_Unit => "??");
```

Changing the Secondary File and Commands

Use the Rci.Edit_Secondary command to alter a secondary file. Note that, if the secondary file is not frozen, you can alter it on the host and you can always alter it on the remote machine, but these methods are not recommended.

Changing Text on the Host

Caution: *Changing the host secondary text file without using the Rci.Edit_Secondary command does not cause demotion of the associated primary Ada unit, nor does it cause a change to the remote copy of the file.*

1. Select the secondary referencer for the secondary unit that you want to edit. If the secondary is controlled, it must be checked out before editing.
2. In a command window, enter `Rci.Edit_Secondary` and press [Promote]. The following actions occur:
   - The primary unit, if coded, is demoted to the installed state.
   - The secondary text file is unfrozen.
   - The secondary text file is opened in a window for editing.
3. Use the editor as described in the *Rational Environment Reference Manual* to enter the non-Ada text into the file. Press [Commit] or use the Common.Commit command periodically to save the file.
4. Close/save the text file.

Caution: *After this procedure is complete, the secondary unit is not refrozen and the primary unit is not repromoted to the coded state. This means that the RCI temporarily has no control over the secondary file.*

Promote the primary to the coded state as soon as feasible to ensure control.

If you change the secondary text file on the host, the RCI downloads and processes it on the remote compilation platform only when you promote its associated primary from the installed to the coded state (in Interactive mode).

Changing Text from the Remote Machine

Usually you should make all changes to source code on the host. If you change the source file on the remote compilation platform, refer to Chapter 5, "Maintaining File Consistency," for utilities to check the consistency of host and remote units and determine which you should update.

To update text changes to an existing host secondary from the remote machine using the Rci.Accept_Remote_Changes command, refer to Chapter 5, "Maintaining File Consistency."

To upload the remote version to the host when a secondary does not already exist:

1. Move to the Units directory of the view into which the remote file should be uploaded, most likely the view in which you will create the secondary file.
2. Enter `Rci.Upload_Unit` in a command window and press **[Complete]**:

```
Rci.Upload_Unit (Remote_Unit_Name    => ">>REMOTE UNIT NAME<<";
                 Into_View           => "<CURSOR>";
                 Upload_To_Text_File => False;
                 Host_Text_File_Name => "";
                 Response            => "<PROFILE>");
```

3. Fill in the parameters as follows:
   - Remote_Unit_Name: Specify the simple name or the full remote name of the source file on the remote machine.
   - Into_View: Change this from the default only if you want to upload the file into some view other than the current view.
   - Upload_To_Text_File: Set to True. This informs the RCI that this is a text file and should not be parsed as an Ada unit.
   - Host_Text_File_Name: Specify a name that is *not* the secondary's filename.
4. Press **[Promote]**.

   The RCI creates the specified text file and places the contents of the text file from the remote machine into it.
5. Use Rci.Create_Secondary with the Host_Text_File_Name as the value of the Secondary_Text parameter to associate the uploaded unit as the secondary text file. (See "Creating the Non-Ada Source File" on page 102.)

Removing Secondary Relationships

Rci.Remove_Secondary removes the secondary referencer and demotes the primary unit to the installed state. It does not delete the secondary text file on the host, and

it has no effect on the remote file. The secondary state file is deleted only after all secondary referencers have been deleted.

Note: *If you delete all secondary referencers and then promote the primary to the coded state, it is treated as any other Ada unit and the RCI downloads and compiles it on the remote compilation platform. You must recompile the primary to make it consistent with its associated remote unit.*

Deleting the Primary or Secondary File

If you unfreeze a secondary text file and then delete it without first using the Rci-.Remove_Secondary command, there is no immediate effect if the primary remains in the coded state. However, if you demote the primary and then repromote it to the coded state, the RCI attempts to find the secondary text file and an error results.

If you delete an Ada primary without first using the Rci.Remove_Secondary command, an error may never be reported, because the secondary text file itself is never referenced during the compilation process.

Changing Secondary Commands and Flags

The RCI provides commands to update the remote operating-system command and Process_Primary flag values for a primary and its connected secondary. Rci.Show-_Secondary displays the current values.

Changing a Secondary's Remote Command

Use Rci.Set_Secondary_Command to associate a command string with a secondary text file.

To change the secondary's remote command:

1. Check out the primary unit.
2. Enter `Rci.Set_Secondary_Command` and press **[Complete]**:

```
Rci.Set_Secondary_Command (Command             := "";
                           Secondary_Referencer := "<CURSOR>";
                           Response            := "<PROFILE>");
```

3. Fill in the parameters:
   - Command: Specify the remote command you want to execute on the remote machine to process the downloaded secondary text file.
   - Secondary_Referencer: Specify the secondary referencer that is associated with the secondary text file.

The primary is demoted to installed.

The next time you promote the primary unit to coded, the RCI downloads the secondary text file and processes it using the remote command.

Setting the Process_Primary Flag

Use Rci.Set_Process_Primary to assign a value to the Process_Primary flag for the primary unit you specify.

To set the Process_Primary flag:

1. Check out the primary unit.
2. Enter `Rci.Set_Process_Primary` and press **[Complete]**:

```
Rci.Set_Process_Primary (Primary_Unit := "<CURSOR>";
                         Value        := False;
                         Response     := "<PROFILE>");
```

3. Fill in the parameters:
   - Primary_Unit: Specify the host Ada unit with which to associate the flag.
   - Value: Specify the value for the Process_Primary flag. If True, the primary unit will be downloaded and processed on the remote machine whenever its secondaries are downloaded and processed.

The primary is demoted to installed.

If Value is True, the next time you promote the primary to the coded state, the RCI downloads the primary with the secondaries to the remote compilation platform and processes it.

Note: *Rci.Create_Secondary also sets the Process_Primary flag. When you perform several Rci.Create_Secondary calls for a given primary, the value of this flag may change with each call.*

PROCESSING SECONDARIES

The RCI downloads a secondary and processes it on the remote compilation platform when you promote its associated primary to the coded state on the host.

Because the secondary is a text file, it has no dependencies of its own. Instead, the RCI uses the primary's compilation dependencies to determine when the secondary is processed.

If the primary is in the coded state, the RCI assumes that the secondary is unchanged. You must demote the primary to at least the installed state and repromote it to cause processing of a changed secondary.

Promoting to Coded

When you promote the primary to the coded state, the following events take place:

1. If the Process_Primary flag is set to True, the RCI downloads the primary and processes it before any of its secondaries. The primary is not downloaded unless the flag is set to True.
2. Each secondary text file is downloaded to the remote compilation platform and processed in the order in which it occurs on the host.
3. Each secondary's associated compiler command is passed to the remote operating system as described in "Creating the Non-Ada Source File" on page 102.
4. Each remote file is then processed according to its passed command. If the command is not successful—for example, if the file is not compiled successfully—error messages are generated and the primary Ada unit is not promoted to the

coded state. See "What Happens During the Coding Step" on page 60 in Chapter 3 for possible errors.

5. If the commands are successful for all of the secondaries, the primary unit is promoted to the coded state and the secondary units are frozen.

Demoting to Installed

Demoting a primary to the installed state has no effect on its secondaries; they remain frozen.

EXAMPLE OF COMPILING NON-ADA CODE

If you promote the Ada specification named Foo to the installed state, it is given a remote-unit name similar to foo_1_s.ada. Since the unit is not promoted to the coded state, nothing is downloaded to the remote machine.

The RCI creates a secondary text file that will contain C code for the remote compilation platform and associates the file with this Ada unit with the command:

```
Rci.Create_Secondary (Primary_Unit    => "Foo";
                      Command         => "cc -c foo.c";
                      Secondary_Text  => "Foo_C";
                      Remote_Name     => "foo.c";
                      Process_Primary => False;
                      Response        => "<PROFILE>");
```

At this time, the secondary text file is created with the name Foo_C and is assigned the remote name foo.c. A secondary referencer is also created and is assigned the name .<Secondary_Foo_C>. The secondary state file is created if one does not already exist:

```
Foo_C                 : File (Text)
Foo'spec              : I Ada
  .<Secondary_Foo_C> : File (Text)
  .<Secondary_State> : File (Text)
```

When you make a request to promote Foo to the coded state, the RCI downloads the text file Foo_C to the remote file foo.c. It then issues the command: cc -c foo.c to the remote operating system. If the target compiler processes the file without errors, the RCI changes Foo's state to coded and freezes Foo_C if it is not already frozen.

If you then execute Rci.Remove_Secondary (Secondary_Referencer => "Foo.<Secondary_Foo_C>"), the RCI demotes Foo to the installed state and deletes .<Secondary_Foo_C>.

Now, if you promote Foo again to the coded state, Foo itself is downloaded to foo_1_s.ada and that file is compiled. This could cause inconsistency if a secondary is later reassociated with Foo.

8 Package Rci

This chapter provides an overview of the functionality of the commands in package Rci. Details on how to use and apply these commands are given in earlier chapters, and those chapters are referenced here.

Following the overview, all of the commands in package Rci are listed in alphabetical order with a description of their parameters and results.

The commands in package Rci are divided into several logical groupings that reflect the association between the commands and the operations that you need to perform while developing in an RCI environment. These groupings are discussed in the following sections.

OPERATIONS FOR BATCH COMPILATION

The following commands provide RCI operations for batch compilation. These commands have an effect only if one of the following is true:

- The extension has set the operation mode to batch through customization default values.
- The extension has been registered with the batch value set in the *Custom_Key*-.Register command.
- The Rci.Operation_Mode library switch for the units is set to "Batch."

In all other cases the batch commands succeed with warning messages.

The batch commands include the following:

- Build_Script: Selects units from the execution closure of units to build on the remote compilation platform and generates a target compilation script to compile these units; optionally downloads units and executes the script.
- Build_Script_Via_Tape: Selects units from the execution closure of units to build on the remote compilation platform and generates an ASCII format tape to contain the units and the target compilation script.
- Execute_Script: Downloads the host script file to the target script file and executes the target compilation script.
- Show_Build_State: Displays the last coding time and the last build time for batch host units.
- Transfer_Units: Transfers all Ada units and any secondaries to the remote machine.
- Upload_Associated_Files: Uploads to the host the associated files that were created on the remote compilation platform.

OPERATIONS FOR CONSISTENCY MANAGEMENT

CMVC ensures that only one user at a time can update a unit joined across many views on the host. However, remote code that is associated with code in RCI views can be updated at will. It is the responsibility of the user to make sure that host and remote source remain consistent. The following operations provide support for management of consistency between host and remote objects:

- Check_Consistency: Checks whether a unit has been edited remotely since it was last coded on the host (and therefore downloaded).
- Accept_Remote_Changes: Uploads changes made remotely into the host unit.
- Upload_Unit: Transfers a file from the remote machine to the host. This is useful when the unit does not currently reside in a host view.
- Upload_Units: Transfers new units, specified in a file, from the remote machine to the host.

OPERATIONS FOR NON-ADA UNITS

The following commands manage the non-Ada source text files that are associated with Ada units:

- Create_Secondary: Creates a non-Ada secondary text unit and associates it with a primary Ada unit.
- Remove_Secondary: Dissociates the indicated secondary and primary units.
- Show_Secondary: Displays the name and associated command instructions of the secondary text unit that is associated with a given Ada primary unit.
- Edit_Secondary: Unfreezes the specified secondary unit and opens it for editing.
- Set_Process_Primary: Sets the Process_Primary flag for the specified primary unit, which controls the processing of the primary on the remote machine.
- Set_Secondary_Command: Associates the specified command string with the specified secondary unit.

OPERATIONS FOR UNITS

The following commands are used to control unit operations on the host and remote compilation platform:

- Link: Used after the coding step has completed to link object modules on the remote machine.
- Set_Remote_Unit_Name: Sets the remote unit name for an Ada unit or secondary unit.
- Show_Remote_Unit_Name: Displays the current remote name of a selected host unit.
- Show_Units: Displays the current information about the state and configuration of the specified units.

OPERATIONS FOR UNIT-COMPILATION OPTIONS

The commands in this group modify and display compiler options on a unit-by-unit basis in an RCI view:

- Set_Unit_Option: Sets compiler options on a unit-by-unit basis. There are two forms of the command, one for options that have arguments and one for options without arguments.
- Display_Unit_Options: Displays information about units' compiler options that were modified by the Set_Unit_Option command.
- Remove_Unit_Option: Disables compiler-option values set by Set_Unit_Option.

OPERATIONS FOR REMOTE LIBRARY MANAGEMENT

The following commands assist in the management of remote libraries. (These operations vary depending on whether your customization has implemented RCI's remote library extensions.)

- Build_Remote_Library: Builds the remote library, which includes both the remote directory and the program library, for an existing RCI view.
- Destroy_Remote_Library: Destroys the remote library associated with an RCI view.
- Display_Default_Naming: Displays the value of Remote_Directory and Remote_Machine for a potential view when you use CMVC view-creation commands and the default switch-naming scheme.
- Rebuild_Remote_Library: Destroys and rebuilds the remote library associated with an RCI view. Use this command when the remote library has been corrupted.
- Refresh_Remote_Imports: Refreshes the remote imports to match the host imports.
- Refresh_View: Initializes an RCI view, optionally preserving the existing information. Use this command when state information may be corrupted or incorrect.
- Show_Remote_Information: Displays the current settings for the remote directory and remote machine that are associated with a view.
- Execute_Remote_Command: Issues a user-specified command on a specified remote machine and redirects the output to an R1000 window on the host.

PROCEDURE ACCEPT_REMOTE_CHANGES

```
procedure Accept_Remote_Changes
         (Unit                  : String  := "<CURSOR>";
          Allow_Demotion        : Boolean := False;
          Compare_Objects       : Boolean := False;
          Remake_Demoted_Units  : Boolean := True;
          Goal                  : Compilation.Unit_State
                                        := Compilation.Coded;
          Response              : String  := "<PROFILE>");
```

Description

Accepts changes from the associated remote units to make the host and remote units consistent.

For Accept_Remote_Changes to be successful, host units that you want to change must be checked in if they are controlled under CMVC and they must be in the source state. They are automatically demoted to the source state if Allow_Demotion is True.

Uploaded units are repromoted to the specified goal state if Remake_Demoted_Units is True.

If Compare_Objects is True, then Accept_Remote_Changes uploads the remote file, compares the original host and the newly uploaded remote files, displays the differences, and accepts changes. If False, it compares units by time, uploads only units whose most recent edit time on the remote machine is later than the download time stored in the state information, and accepts the changes.

For detailed information on determining the consistency between the host and remote units and when and how to use this command, see "Consistency between Host and Remote Units" on page 78 in Chapter 5.

Parameters

`Unit : String := "<CURSOR>";`

Specifies a naming expression that describes one or more host Ada units in a single RCI view or the entire RCI view. These can also be primary units with corresponding secondaries.

`Allow_Demotion : Boolean := False;`

Specifies whether units on the host that need to be demoted to the source state in order to be updated should be demoted. If False, uploading fails where a host unit cannot be demoted to the source state.

`Compare_Objects : Boolean := False;`

Specifies the method for consistency checking. If True, consistency is checked by comparing host and remote objects rather than timestamps.

Remake_Demoted_Units : Boolean := True;

Specifies whether host units that are demoted as a result of Accept_Remote_Changes should be promoted to the state indicated by the Goal parameter.

Goal : Compilation.Unit_State := Compilation.Coded;

Specifies the state to which host units demoted during Accept_Remote_Changes should be promoted if Remake_Demoted_Units is True. Useful values are Compilation.Source and Compilation.Installed; Compilation.Coded generates a warning that units will be promoted only to the installed state.

Response : String := "<PROFILE>";

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Restrictions

With a Telnet customization of RCI, the Compare_Objects parameter is ignored and True is used as its value.

References

"Consistency Between Host And Remote Units," page 76

PROCEDURE BUILD_REMOTE_LIBRARY

```
procedure Build_Remote_Library
         (View             : String := "<CURSOR>";
          Remote_Machine   : String := "<DEFAULT>";
          Remote_Directory : String := "<DEFAULT>";
          Response         : String := "<PROFILE>");
```

Description

Builds the remote library for an existing RCI view.

The remote library consists of both the remote directory, where source is stored, and the program library into which it is compiled.

If <DEFAULT> is given for the Remote_Machine and Remote_Directory parameters, the values are taken from the view's compiler switches. If these values do not exist in the view's compiler switches, they must be provided here.

This routine assumes that the Make_Path_Postprocess RCI extension has been implemented. This extension may or may not be implemented from one RCI customization to another.

If the Import_Preprocess and Import_Postprocess extensions are enabled, the remote imports are created by this routine.

Parameters

`View : String := "<CURSOR>";`

Specifies a naming expression for the existing RCI combined view for which to create the associated remote library.

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

Specifies the remote machine name on which to create the remote library.

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

Specifies the directory name on the remote machine where the remote library is created.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Building a Remote Library," page 88
procedure Destroy_Remote_Library

PROCEDURE BUILD_SCRIPT

```
procedure Build_Script
         (Host_Units             :String   := "<IMAGE>";
          Link_Main_Units        :Boolean  := True;
          Transfer_To_Target     :Boolean  := True;
          Host_Script_File       :String   := "<DEFAULT>";
          Remote_Script_File     :String
                                       := ">> FULL REMOTE NAME <<";
          Build_List_File        :String   := "<DEFAULT>";
          Execute_Script         :Boolean  := False;
          Effort_Only            :Boolean  := False;
          Minimal_Recompilation  :Boolean  := True;
          Make_Units             :Boolean  := False;
          Response               :String   := "<PROFILE>");
```

Description

Selects units from the execution closure of Host_Units that need to be built on the target, generates a script file on the host to compile these units on the target, and then downloads the host script file to the remote script file.

Only obsolete units—that is, coded units whose coding times are more recent than their build times—are selected for the build. Once a unit is entered into the compilation script, it is considered to have been built.

If Transfer_To_Target is True, the units selected for the build are also downloaded before the script is downloaded, if the host and remote compilation platform can communicate through FTP.

If Execute_Script is True, the remote script file is then executed on the target.

If attempts to make a connection to the remote machine associated with the current view fail, correct the problem and then retry downloading and executing the existing script using the Rci.Execute_Script command.

Parameters

`Host_Units : String := "<IMAGE>";`

Specifies a naming expression for the units (and their execution closure) to be selected for a build (compiling and optionally linking remotely). All units in the execution closure of Host_Units that have been coded since their last build will be selected for the current build.

Host_Units can also be an indirect file, such as the Build_List_File specified below, containing the complete (including execution closure) list of host units to be built on the target. The batch script is then generated from this list of units. Host_Units must not specify units belonging to different targets. Also Host_Units must belong to views associated with only one remote machine.

Link_Main_Units : Boolean := True;

Specifies whether to include link commands in the script file for main units. If True, link commands are included for all main units (units containing a pragma Main) specified by Host_Units.

Transfer_To_Target : Boolean := True;

Determines whether units are downloaded to the compilation platform as the script is built. If True, units selected for the build are also downloaded to the target. This downloads units whether or not they have changed since the last download. This option has no effect if Effort_Only is True.

Host_Script_File : String := "<DEFAULT>";

Specifies the name of the batch script file to create on the host R1000 for a particular target. The default filename is Batch_Script. If the default is used, the command builds a file named Batch_Script in the current context. The command does not overwrite any existing script file and fails if it finds one.

Remote_Script_File : String := ">> FULL REMOTE NAME <<";

Specifies the name of the remote file into which the Host_Script will be downloaded. Failure to download will cause warnings to be generated, but the build command will still succeed. The Remote_Script_File must specify the complete pathname on the remote machine.

Build_List_File : String := "<DEFAULT>";

Specifies the name for a new host file in which to place a list of the names of the host units that were selected for remote compilation during the current build. The default filename is Units_To_Build; if the default is used, the file is created in the current context. This file is created when the Build_Script command begins execution; if the build fails, the file remains empty. Failure to create this file causes the build to fail. Units are listed in Units_To_Build in the order in which they need to be compiled. This file provides information when regenerating a Host_Script_File to upload the associated files for a particular build.

Execute_Script: Boolean := False;

Specifies whether to execute the remote script file after its creation. If True, the script is automatically executed on the target with output being redirected to an R1000 window. Failure to execute this script will produce warnings but will not cause the build to fail. The Build_Script command will fail if the remote execution fails. However, units that participated in the build will still be considered to have been built.

Effort_Only : Boolean := False;

Specifies, if True, that a list be displayed of the units specified by Host_Units that would be included in the build without performing any other build operations.

Minimal_Recompilation : Boolean := True;

Specifies whether to include only obsolete units in the build script. If True, only specified units (and units in their execution closure) that have been made codable since the last batch script was generated are included in the script. If False, all units in the execution closure of the specified units are included in the script.

```
Make_Units : Boolean := False;
```

Specifies, if True, that any units in the closure of Host_Units that are in the source or installed state be promoted to the coded state on the host before the batch script is generated.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Building Batch Scripts for Networked Environments," page 68
procedure Execute_Script

PROCEDURE BUILD_SCRIPT_VIA_TAPE

```
procedure Build_Script_Via_Tape
         (Host_Units             : String  := "<IMAGE>";
          Link_Main_Units        : Boolean := True;
          Host_Script_File       : String  := "<DEFAULT>";
          Remote_Script_File     : String
                                        := ">> FULL REMOTE NAME <<";
          Build_List_File        : String  := "<DEFAULT>";
          Format                 : String  := "R1000";
          Volume                 : String  := "";
          Label                  : String  := "rci_build";
          Logical_Device         : String  := "rci";
          Effort_Only            : Boolean := False;
          Minimal_Recompilation  : Boolean := True;
          Make_Units             : Boolean := False;
          Response               : String  := "<PROFILE>");
```

Description

Selects units from the execution closure of Host_Units that need to be built on the target, generates the batch-compilation script to compile these units, and generates a tape in ANSI format to contain the batch-compilation script, the units to be built and a script to move the units to the right place on the target.

Only coded units whose coding times are more recent than their build times are selected for the build. Once a unit is entered into the batch-compilation script, it is considered to have been built.

Parameters

`Host_Units : String := "<IMAGE>";`

`Link_Main_Units : Boolean := True;`

`Host_Script_File : String := "<DEFAULT>";`

See procedure Build_Script for descriptions of the parameters above.

`Remote_Script_File : String := ">> FULL REMOTE NAME <<";`

Specifies the name of the remote file to use when copying the Host_Script onto the tape. Failure to access the tape unit will cause warnings to be generated, but the build command will still succeed. The Remote_Script_File parameter must specify the complete pathname on the remote machine.

`Build_List_File : String := "<DEFAULT>";`

See procedure Build_Script for a description of this parameter.

`Format : String := "R1000";`

Specifies the type of tape to be written—Ansi, R1000, or R1000_Long. Refer to the documentation on the Archive package for more information on Format options.

`Volume : String := "";`

Specifies the tape volume name.

```
Label : String := "rci_build";
```

Specifies a string written in front of the data on the tape.

```
Logical_Device : String := "rci";
```

Specifies the logical device name for the tape device on the remote machine. The move script references the logical device rather than the physical device name.

```
Effort_Only : Boolean := False;
```

```
Minimal_Recompilation : Boolean := True;
```

```
Make_Units : Boolean := False;
```

See procedure Build_Script for descriptions of these parameters.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Building Batch Scripts for Tape Environments," page 70
package Archive in the Library Management (LM) book of the *Rational Environment Reference Manual*

PROCEDURE CHECK_CONSISTENCY

```
procedure Check_Consistency
         (Unit            : String   := "<CURSOR>";
          Compare_Objects : Boolean  := False;
          Response        : String   := "<PROFILE>");
```

Description

Checks the consistency of units against their associated units in the corresponding remote directory, when given a naming expression describing one or more units from a single view.

The method of consistency checking is controlled by the Compare_Objects parameter value, either by comparing the remote and host objects or by comparing their edit and download timestamps.

Each time that a host unit is downloaded to the remote machine, the host unit's download time is set in the RCI state information by retrieving the edit time of the remote text file. A host unit is considered inconsistent with its remote unit if the download time of the host unit differs from the edit time of the remote unit when the Check_Consistency command is executed.

If Compare_Objects is True, the remote unit is uploaded and then compared line by line to the host object for consistency.

This command also lists units that are registered in the state information but for which no remote files exist.

This command does not check whether the host unit has changed since it was last downloaded, nor does it check whether the remote unit has become obsolete in the target-compiler program library.

Parameters

`Unit : String := "<CURSOR>";`

Specifies a naming expression describing one or more units in a single view. The units can be primary units with secondaries. The units are compared with their associated remote files, whose names can be viewed with the Rci.Show_Remote_Unit-_Name command.

`Compare_Objects : Boolean := False;`

Specifies the method for consistency checking. If True, consistency is checked by comparing host and remote objects rather than timestamps.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Restrictions

For Telnet customizations of the RCI, the Compare_Objects parameter is ignored and True is used as its value.

References

"Determining Consistency of Host and Remote Units," page 78

PROCEDURE COLLAPSE_SECONDARY_REFERENCERS

```
procedure Collapse_Secondary_Referencers
         (Directory : String  := "<CURSOR>";
          Response  : String  := "<PROFILE>");
```

Description

Collapses text files within the specified directory and the subdirectories below it to secondary-referencer (pointy) files.

In some versions, Archive.Copy does not copy subobjects such as secondary-referencer files. Before you copy a view containing secondaries to a new location, run Rci.Expand_Secondary_Referencers to expand secondary-referencer files into a form that can be copied. Once your copy operation has completed, run Rci.Collapse_Secondary_Referencer to restore the secondary-referencer subobjects.

Since Archive.Copy cannot copy secondary referencers:

1. Convert secondary referencers to text files using the Rci.Expand_Secondary-_Referencers command.
2. Archive.Copy these text files.
3. Convert text files back to secondary referencers using this command.

Parameters

`Directory : String := "<CURSOR>";`

Specifies the directory that contains the files to collapse.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

PROCEDURE CREATE_SECONDARY

```
procedure Create_Secondary
         (Primary_Unit     : String  := "<CURSOR>";
          Command          : String  := "";
          Secondary_Text   : String  := "";
          Remote_Name      : String  := "<DEFAULT>";
          Process_Primary  : Boolean := False;
          Response         : String  := "<PROFILE>");
```

Description

Creates a relationship between the specified non-Ada secondary text and the specified Ada primary unit through a secondary referencer.

This is the method that the RCI uses to support non-Ada units. Create_Secondary associates a specified secondary text file with an Ada primary unit. The secondary text file is created if it does not already exist. A primary can be associated with more than one secondary. The RCI uses the primary's dependencies to determine the compilation ordering for the associated secondary files.

The secondary text file contains the non-Ada source code. If the secondary text file does not already exist, Create_Secondary creates that text file in the same view as the primary unit.

The primary unit points to the secondary text file using a pointy file called the *secondary referencer.* The name of the secondary referencer is of the form Primary_Unit_Name.<Secondary_*Secondary_Text_File_Name*>.

The secondary-referencer file contains the following information:

- Name of the secondary text file associated with the primary
- Remote name into which the secondary text unit is to be downloaded
- Remote command to be executed after the secondary text is downloaded
- Download time of the secondary text unit on the remote machine

If the primary unit is in the coded state, it will be demoted to installed before its secondary referencer is created.

The primary unit must be checked out.

When Create_Secondary associates the first secondary file with a primary unit, it creates a *secondary state file* for the primary unit. This file contains the current status of the Process_Primary parameter, which controls how the primary unit is handled when it is promoted to coded in the host environment. If the Process_Primary value is True, the primary unit is downloaded and processed on the remote machine with the secondaries. If it is False, the primary unit is not downloaded. The value in the secondary state file reflects the value of the most recent call to Create_Secondary. For a primary unit with more than one secondary, the state file contains the last value used for the Process_Primary parameter. You can alter that value by using the Set_Process_Primary command.

Whenever a primary with an associated secondary is promoted to the coded state, the RCI sends the secondary text file to the remote machine. The remote compilation system uses the command associated with the secondary to process the secondary.

The primary is downloaded and processed (before its secondaries) only if the Process_Primary parameter associated with it is set to True.

If you then edit the secondary using the Edit_Secondary command on the host environment, the associated primary is first demoted to the installed state. This means that the changed secondary unit is downloaded to the remote machine and compiled there when the primary is recoded.

To make inadvertent changes more difficult, secondary text units are frozen in the host library whenever their primary is promoted to the coded state.

All secondary commands operate on the secondary-referencer file instead of the secondary text file.

The Process_Primary flag contains the value from the last Create_Secondary call. You may need to change this value when working with multiple secondaries.

Parameters

`Primary_Unit : String := "<CURSOR>";`

Specifies the name of a single Ada unit with which the secondary unit should be associated.

`Command : String := "";`

Specifies the remote operating-system command that processes the secondary text file on the remote machine.

`Secondary_Text : String := "";`

Specifies the name of a text file in the same view as the primary unit with which the Ada primary unit should be associated. This text file usually contains source code in some language other than Ada.

`Remote_Name : String := "<DEFAULT>";`

Specifies the remote name into which the secondary text unit is downloaded when the primary is promoted to coded in the host environment. The default is the same name as in the host environment.

`Process_Primary : Boolean := False;`

Specifies, if True, that the primary unit be downloaded and processed (before any secondaries are processed) when it is promoted to coded in the host environment. If False, the primary unit is not downloaded and processed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Restrictions

The following restrictions apply:

- A secondary text file cannot be associated with more than one primary.
- An Ada subunit cannot be a primary unit.
- Both units must be in the same view.

Examples

The following command creates a secondary:

```
Create_Secondary (Primary_Unit     => "tulip'body",
                  Command          => "cc -c open.c",
                  Secondary_Text   => "open_c";
                  Remote_Name      => "open.c";
                  Process_Primary  => False);
```

This results in the following library structure:

```
Open_C                : File (Text)
Tulip'Body            : I Ada -Primary
  .<Secondary_Open_C> : File (Text)-- Secondary referencer
  .<Secondary_State>  : File (Text)-- Secondary state
```

When Tulip'Body is promoted to the coded state, the secondary text file Open_C is downloaded into open.c on the remote machine and the target compiler executes the command cc -c open.c.

References

"Removing Remote Libraries," page 91
procedure Edit_Secondary
procedure Remove_Secondary
procedure Set_Process_Primary
procedure Set_Secondary_Command
procedure Show_Secondary

PROCEDURE DESTROY_REMOTE_LIBRARY

```
procedure Destroy_Remote_Library
         (View     : String := "<CURSOR>";
          Response : String := "<PROFILE>");
```

Description

Destroys the remote library associated with the view.

This command destroys the remote program library, Ada source, and working directory and demotes units on the host.

This routine assumes that the Destroy_View_Postprocess RCI extension has been implemented. This extension may or may not be implemented from one RCI customization to another.

Parameters

`View : String := "<CURSOR>";`

Specifies the name of the RCI view associated with the remote library.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Removing Remote Libraries," page 91
procedure Build_Remote_Library
procedure Rebuild_Remote_Library

PROCEDURE DISPLAY_DEFAULT_NAMING

```
procedure Display_Default_Naming
        (Potential_View : String := "";
         Target_Key     : String := "";
         Response       : String := "<PROFILE>");
```

Description

Displays the remote machine and directory names that would be chosen by CMVC if Potential_View were to be created with the given Target_Key.

This allows you to determine whether you need to explicitly specify the Remote_Machine and Remote_Directory parameters by using commands in package Rci_Cmvc.

Parameters

`Potential_View : String := "";`

Specifies the complete name of a nonexistent view. For example, the Rci_Cmvc.Initial command allows you to specify only the base name of the working view (for example, Rev1), to which the command would append the text _Working, resulting in a *complete* view name of Rev1_Working.

`Target_Key : String := "";`

Specifies the target key under which the view would be created.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Displaying Defaults for Remote RCI Names," page 36

PROCEDURE DISPLAY_UNIT_OPTIONS

```
procedure Display_Unit_Options
         (Units    : String := "<CURSOR>";
          Response : String := "<PROFILE>");
```

Description

Displays, for each unit specified, the compiler options enabled for that unit by the Set_Unit_Option command.

These values override the current values in the compiler-options switches of the library-switch files.

Parameters

`Units : String := "<CURSOR>";`

Specifies a naming expression for one or more host Ada units for which enabled unit options are displayed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Specifying Unit-Specific Compiler Options," page 51
procedure Remove_Unit_Option
procedure Set_Unit_Option

PROCEDURE EDIT_SECONDARY

```
procedure Edit_Secondary
        (Secondary_Referencer : String  := "<CURSOR>";
         In_Place             : Boolean := False;
         Visible              : Boolean := True;
         Response             : String  := "<PROFILE>");
```

Description

Allows editing of the secondary unit that is associated with a primary unit through the secondary referencer.

This command opens the secondary text file on the host environment for editing and displays it in a window on the screen.

If the primary is coded, it is demoted to installed before editing begins. The secondary text file is unfrozen for editing. When the primary is promoted to coded, the secondary is downloaded, processed, and refrozen in the host environment.

Parameters

`Secondary_Referencer : String := "<CURSOR>";`

Specifies the secondary-referencer file that is associated with the secondary text file that is to be edited.

`In_Place : Boolean := False;`

Specifies, if True, that the secondary associated with the specified primary is opened in a window that replaces the window in which the cursor is currently located. If False, the default, a different window is used.

`Visible : Boolean := True;`

Ignored.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Errors

This command returns an error if you try to edit these units:

- An Ada unit that does not have a secondary
- A text unit that is not a secondary
- Any type of file other than an Ada unit or text file

References

"Changing the Secondary File and Commands," page 102
procedure Create_Secondary
procedure Remove_Secondary
procedure Show_Secondary

PROCEDURE EXECUTE_REMOTE_COMMAND

```
procedure Execute_Remote_Command
         (Remote_Command   : String := "";
          Remote_Machine   : String := "<DEFAULT>";
          Remote_Username  : String := "<DEFAULT>";
          Remote_Password  : String := "<DEFAULT>";
          Remote_Directory : String := "<DEFAULT>";
          The_Key          : String := "<DEFAULT>";
          Response         : String := "<PROFILE>");
```

Description

Executes the specified remote command on the specified remote machine.

The output of the remote command is displayed in a host-environment window.

Parameters

Remote_Command : String := "";

Specifies the remote operating-system command to be executed on the remote platform.

Remote_Machine : String := "<DEFAULT>";

Specifies the remote machine on which to execute the command. The default is taken from the enclosing library-switch file or the session-switch file if not defined in the library switches.

Remote_Username : String := "<DEFAULT>";

Specifies the username to use for login on the remote machine. The default is taken from the enclosing library-switch file, the session-switch file, or the Remote_Passwords file if not defined in the library switches.

Remote_Password : String := "<DEFAULT>";

Specifies the password to use for login on the remote machine. The default is taken from the enclosing library-switch file, the session-switch file, or the Remote_Passwords file if not defined in the library switches.

Remote_Directory : String := "<DEFAULT>";

Specifies the remote directory from which to execute the command. The default is taken from the enclosing library-switch file or the session-switch file if not defined in the library switches.

The_Key : String := "<DEFAULT>";

Specifies a valid RCI target key. The default is taken from the target key of the enclosing world.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

PROCEDURE EXECUTE_SCRIPT

```
procedure Execute_Script
   (Host_Script_File         : String  := "<IMAGE>";
    Remote_Script_File       : String  := ">> FULL REMOTE NAME <<";
    Remote_Machine           : String  := "<DEFAULT>";
    Remote_Username          : String  := "<DEFAULT>";
    Remote_Password          : String  := "<DEFAULT>";
    Remote_Directory         : String  := "";
    Effort_Only              : Boolean := False;
    Display_Remote_Commands  : Boolean := False;
    The_Key                  : String  := "<DEFAULT>";
    Response                 : String  := "<PROFILE>");
```

Description

Downloads the specified host script file to the specified remote machine and executes the script in cases where the host and target can communicate through FTP and Telnet.

Script files are built by the Rci.Build_Script command. They can be executed at that time by setting the Execute_Script parameter on the Build_Script command to True. In cases where the downloading or execution failed during the build, where the Execute_Script parameter was False, or where the same script file must be executed on several remote machines (as described in Transfer_Units), this command can be used to download and execute existing batch scripts.

Parameters

Host_Script_File : String := "<IMAGE>";

Specifies the name of a host script file generated by the Build_Script or the Build_Script_Via_Tape command that is to be downloaded to the specified remote machine.

Remote_Script_File : String := ">> FULL REMOTE NAME <<";

Specifies the name of the remote file into which the Host_Script will be downloaded. The Remote_Script_File must specify the complete pathname on the remote machine.

Remote_Machine : String := "<DEFAULT>";

Remote_Username : String := "<DEFAULT>";

Remote_Password : String := "<DEFAULT>";

Remote_Directory : String := "";

Specify the information about the remote machine needed to download the script file. The Display_Default_Naming command can be used to determine the default values for Remote_Machine and Remote_Directory.

Effort_Only : Boolean := False;

Specifies, if True, that the name of the script file be displayed, but it is not downloaded or executed.

`Display_Remote_Commands : Boolean := False;`

Specifies whether to display the remote script-file commands on the host as they execute on the remote machine. The default, False, prevents the commands from being displayed.

`The_Key : String := "<DEFAULT>";`

Specifies a valid RCI target key to use. The default is the target key of the enclosing world.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Executing a Batch Script on the Compilation Platform," page 71
procedure Build_Script

PROCEDURE EXPAND_SECONDARY_REFERENCERS

```
procedure Expand_Secondary_Referencers
         (Directory : String := "<CURSOR>";
          Response  : String := "<PROFILE>");
```

Description

Expands out secondary-referencer (pointy) files to text files in the specified directory and all subdirectories within it.

In some versions, Archive.Copy does not copy subobjects such as secondary-referencer files. Before you copy a view containing secondaries to a new location, run Rci.Expand_Secondary_Referencers to expand secondary-referencer files into a form that can be copied. Once your copy operation has completed, run Rci.Collapse_Secondary_Referencer to restore the secondary-referencer subobjects.

Since Archive.Copy cannot copy secondary referencers:

1. Convert all secondary referencers to text files using this command.
2. Archive.Copy these text files.
3. Convert text files back to secondary referencers using the Collapse_Secondary-_Referencers command.

Parameters

`Directory : String := "<CURSOR>";`

Specifies the directory that contains the files to expand.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

PROCEDURE LINK

```
procedure Link
         (Main_Unit          : String  := "<CURSOR>";
          Make_Uncoded_Units : Boolean := False;
          Executable_Name    : String  := "";
          Response           : String  := "<PROFILE>");
```

Description

Links main units using the remote linker.

Each main unit and all units in its closure must be successfully promoted to the coded state before the link can take place; setting the Make_Uncoded_Units parameter to True requests that all uncoded units in the main unit's closure be coded before the link operation is executed.

Parameters

`Main_Unit : String := "<CURSOR>";`

Specifies a naming expression for one or more host Ada main units whose remote counterparts are to be linked with all appropriate object modules on the remote machine.

`Make_Uncoded_Units : Boolean := False;`

Specifies whether the main unit and units in its closure that are currently not in the coded state should be promoted to the coded state before the link takes place. If False, the link fails unless all units are already in the coded state. If True, then normal compilation requirements must be met, and coding takes place as described in Chapter 3, "Getting Started"

`Executable_Name : String := "";`

Specifies the name to use for the executable module that is the output from the link operation on the remote machine. If the null string is specified, then the default name is determined as described in "Remote Files and Names" on page 56 in Chapter 3. Only one main unit can be linked at a time if a nonnull Executable_Name is specified.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"What Happens During the Linking Step," page 61

PROCEDURE REBUILD_REMOTE_LIBRARY

```
procedure Rebuild_Remote_Library
         (View                 : String  := "<CURSOR>";
          Remake_Demoted_Units : Boolean := True;
          Remote_Machine       : String  := "<DEFAULT>";
          Remote_Directory     : String  := "<DEFAULT>";
          Response             : String  := "<PROFILE>");
```

Description

Rebuilds the associated remote libraries for one or more RCI views.

This command carries out the following operations for each specified view:

- Demotes all units to the installed state
- Destroys the current remote library
- Rebuilds the remote library
- Recodes all units

Parameters

`View : String := "<CURSOR>";`

Specifies a naming expression for the existing RCI combined view for which to destroy and then rebuild the associated remote library.

`Remake_Demoted_Units : Boolean := True;`

Specifies, if True, that all units and dependents are to be recoded in the rebuilt library.

`Remote_Machine : String := "<DEFAULT>")`

Specifies the name of the remote machine name on which to create the remote library.

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

Specifies the directory name on the remote machine where the new remote library will be created.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Rebuilding an Existing Remote Library," page 88

PROCEDURE REFRESH_REMOTE_IMPORTS

```
procedure Refresh_Remote_Imports
          (View     : String := "<CURSOR>";
           Response : String := "<PROFILE>");
```

Description

Compares and updates remote imports to match host imports.

This operation is described in more detail in Chapter 6, "Library Management" This command directly calls Import_Preprocess and Import_Postprocess if these routines are implemented in your customization.

Parameters

`View : String := "<CURSOR>";`

Specifies a naming expression for an RCI combined view whose imports should be compared to the remote imports and used to update mismatched remote imports.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Imports," page 92

PROCEDURE REFRESH_VIEW

```
procedure Refresh_View
         (View           : String  := "<CURSOR>";
          Retain_History : Boolean := True;
          Response       : String  := "<PROFILE>");
```

Description

Refreshes RCI state information that is believed to be corrupted or inconsistent.

If the Retain_History parameter is True, the following information is retained from the existing RCI state information:

- Remote unit names
- Download timestamps
- Unit options

If Retain_History is False, the command completely reinitializes the state information for the specified view.

Parameters

`View : String := "<CURSOR>";`

Specifies an RCI combined view. This must be an RCI view.

`Retain_History : Boolean := True;`

Specifies whether to retain existing history.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual*.

Errors

This command results in errors if the view is not an RCI view.

PROCEDURE REMOVE_SECONDARY

```
procedure Remove_Secondary
         (Secondary_Referencer : String := "<CURSOR>";
          Response             : String := "<PROFILE>");
```

Description

Removes the specified secondary referencer, thus breaking the association between a primary and a secondary unit.

This command:

- Demotes the associated primary unit from the coded state (if that is its current state) to the installed state.
- Removes the secondary referencer.
- Leaves unchanged the secondary text file previously associated with the primary.
- Makes no changes on the remote machine; in particular, the copy of the secondary unit is not removed from the remote machine. This means that the primary unit must be recompiled to make its associated remote unit consistent with the host unit.

The primary unit must be checked out.

Parameters

`Secondary_Referencer : String := "<CURSOR>";`

Specifies a naming expression for a single secondary-referencer file that is associated with a secondary text file.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

Chapter 7, "Using Non-Ada Code with the RCI"
procedure Create_Secondary
procedure Show_Secondary

PROCEDURE REMOVE_UNIT_OPTION

```
procedure Remove_Unit_Option
         (Option_Switch : String := "<ALL>";
          Units         : String := "<CURSOR>";
          Response      : String := "<PROFILE>");
```

Description

Disables, for each unit specified, the specified option set by the Set_Unit_Option command.

Switch values for these options are reset to the values in the associated compiler switch files.

Parameters

`Option_Switch : String := "<ALL>";`

Specifies the name of the compiler-option switch as it appears in the library-switch file. This can be either the complete switch filename (Rci.*Custom_Key_Option*) or the compiler-option switch name (*Option*).

`Units : String := "<CURSOR>";`

Specifies a naming expression for the units associated with the unit switch options.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Specifying Unit-Specific Compiler Options," page 51
procedure Display_Unit_Options
procedure Set_Unit_Option

PROCEDURE SET_PROCESS_PRIMARY

```
procedure Set_Process_Primary
        (Primary_Unit : String  := "<CURSOR>";
         Value        : Boolean := False;
         Response     : String  := "<PROFILE>");
```

Description

Sets the Process_Primary flag for the specified primary unit.

This operation controls the handling of the specified primary unit when it is promoted to coded. In Interactive mode, if the Process_Primary flag is True, the primary unit is downloaded and processed on the remote machine before its secondary units are processed. If False, the primary unit is not downloaded with the secondary units.

The primary unit must be checked out.

The primary unit is demoted to the installed state if necessary.

Parameters

`Primary_Unit : String := "<CURSOR>";`

Specifies the host Ada unit with which to associate the flag.

`Value : Boolean := False;`

Specifies the value of the Process_Primary flag. If True, this causes the primary unit to be downloaded and processed on the remote machine when it is promoted to coded in the host environment.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Setting the Process_Primary Flag," page 104

PROCEDURE SET_REMOTE_UNIT_NAME

```
procedure Set_Remote_Unit_Name
          (Remote_Name     : String  := ">>REMOTE NAME<<";
           Unit            : String  := "<CURSOR>";
           Allow_Demotion  : Boolean := True;
           Response        : String  := "<PROFILE>");
```

Description

Specifies the name to use when the host unit, either an Ada unit or a secondary text file, is downloaded to the remote machine.

This command normally is not needed, because names are created automatically based on your RCI customization. Remote names are created automatically for Ada units. The name of a secondary text file is specified in the Create_Secondary command, or the RCI uses the same name as in the host environment. For more information, see "Remote Files and Names" on page 56 in Chapter 3.

When an Ada unit is coded, the unit is downloaded into a remote unit with the specified name. For a secondary referencer, the secondary text file associated with the primary is downloaded into the named remote unit.

The user must ensure that the remote name is unique; the RCI does not verify whether the remote name is used elsewhere, and a duplicate remote name could result in the unintentional overwriting of another remote file.

The current remote names for host units can be displayed with the Show_Remote-_Unit_Name or Show_Units command.

If a file exists on the remote machine with an old remote-unit name, you should delete or rename this file, because the RCI does not perform this cleanup operation.

If the specified host unit is in the coded state, it is demoted to the installed state when its remote-unit name is changed. This ensures that it will be recoded at some later date, thus creating the appropriately named remote file. This can be circumvented by setting the Allow_Demotion parameter to False.

For secondaries, the primary unit must be checked out.

Parameters

`Remote_Name : String := ">>REMOTE NAME<<";`

Specifies the name that will be used when the unit is downloaded to the remote machine. This name is stored permanently as part of the RCI state information. It can be a maximum of the customization-specified number of characters. If this parameter is the null string, the unit's default remote name is assigned or restored as described above.

`Unit : String := "<CURSOR>";`

Specifies a naming expression for a single unit in an RCI view whose remote name should be set to the indicated Remote_Name. This must specify an Ada unit or secondary-referencer name. If Unit is an Ada unit, the unit, when downloaded, goes

into a remote unit with the specified remote-unit name. If Unit specifies a secondary referencer, the associated secondary text file is downloaded into the remote-named file.

`Allow_Demotion : Boolean := True;`

Specifies whether the host unit should be demoted to the installed state. If this parameter is set to False, the host unit remains in its current state. If the host unit is coded and Allow_Demotion is set to False, the remote name is not changed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Remote Files and Names," page 56

PROCEDURE SET_SECONDARY_COMMAND

```
procedure Set_Secondary_Command
         (Command              : String := "";
          Secondary_Referencer : String := "<CURSOR>";
          Response             : String := "<PROFILE>");
```

Description

Assigns the given command string to a secondary.

This command executes on the remote machine to process the secondary text file after it is downloaded to the remote machine (when its primary is coded). The secondary referencer indicates which secondary text file is associated with the command.

The primary unit must be checked out.

The primary unit is demoted to the installed state if necessary.

Parameters

`Command : String := "";`

Specifies the command to execute on the remote compilation platform when processing the downloaded secondary text file.

`Secondary_Referencer : String := "<CURSOR>";`

Specifies the secondary referencer for the secondary text file with which to associate the command.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Changing a Secondary's Remote Command," page 104

PROCEDURE SET_UNIT_OPTION

```
procedure Set_Unit_Option
         (Option_Switch : String  := ">>OPTION<<";
          Switch_Value  : Boolean;
          Units         : String  := "<CURSOR>";
          Response      : String  := "<PROFILE>");

procedure Set_Unit_Option
         (Option_Switch : String  := ">>OPTION<<";
          Switch_Value  : String;
          Units         : String  := "<CURSOR>";
          Response      : String  := "<PROFILE>");
```

Description

Assigns compiler options on a unit-by-unit basis, overriding what appears in the switch file.

The RCI ignores the value in the switch file for units that have that option set. The option value stays enabled until it is removed with Remove_Unit_Option.

Parameters

`Option_Switch : String := ">>OPTION<<";`

Specifies the switch-name entry in the option customization. This can be either the fully qualified switch filename or the option-switch name.

`Switch_Value : Boolean;`

Specifies the value for the associated compiler switch.

`Switch_Value : String;`

Specifies the argument value to assign to the associated compiler switch.

`Units : String := "<CURSOR>";`

Specifies a naming expression for the units whose unit switch options are to be set.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Specifying Unit-Specific Compiler Options," page 51
procedure Display_Unit_Options
procedure Remove_Unit_Option

PROCEDURE SHOW_BUILD_STATE

```
procedure Show_Build_State
         (Host_Units           : String  := "<CURSOR>";
          Execution_Closure    : Boolean := False;
          Obsolete_Units_Only  : Boolean := False;
          Response             : String  := "<PROFILE>");
```

Description

Displays the last coding time and the last build time for the specified units.

If you are operating in batch mode as described in "Operations for Batch Compilation" on page 109, the coded state indicates only that dependencies have been recorded for a unit and its closure, not necessarily that the unit has been downloaded and compiled on the remote machine. In this mode, the Build_Script command must be run to generate a script, and then the script must be executed on the remote compilation server. The Show_Build_State command displays a report that lists the coding times and the build times for the specified units; you can use the Obsolete-_Units_Only parameter to include only the units that are out of date; that is, have not been generated in a script since they were last promoted to the coded state.

The output of this command appears as follows:

```
unit : !Proj.Sub.Rev1_Working.Units.A'Body
last coding time : 92/04/06 09:47:31
last build  time : 92/04/06 08:47:31
```

Parameters

`Host_Units : String := "<CURSOR>";`

Specifies a set of units (and optionally their execution closures) whose build states are to be displayed. Only coded units are reported.

`Execution_Closure : Boolean := False;`

Specifies whether to also display information about coded units in the execution closure of Host_Units. If True, they are included in the display.

`Obsolete_Units_Only : Boolean := False;`

Specifies, if True, that the state of only obsolete units (units whose coding time is more recent than their build time) are reported. If False, all units specified by Host_Units and Execution_Closure are displayed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Checking the Build State," page 71
procedure Build_Script
procedure Build_Script_Via_Tape

PROCEDURE SHOW_REMOTE_INFORMATION

```
procedure Show_Remote_Information
         (View     : String := "<CURSOR>";
          Response : String := "<PROFILE>");
```

Description

Displays the names of the remote machine, remote directory, and remote program library associated with the specified host view.

Since this command simply displays the values of the Ftp.Remote_Directory switch and the Ftp.Remote_Machine switch, the same effect is achieved by displaying the Ftp.Remote_Directory and Ftp.Remote_Machine switches in the view's Compiler-_Switches library-switch or session-switch file.

Parameters

`View : String := "<CURSOR>";`

Specifies the RCI view whose Ftp.Remote_Directory and Ftp.Remote_Machine values should be displayed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Specifying Remote Login Information," page 20

PROCEDURE SHOW_REMOTE_UNIT_NAME

```
procedure Show_Remote_Unit_Name
         (Unit     : String := "<CURSOR>";
          Response : String := "<PROFILE>");
```

Description

Displays the remote name for the specified unit.

For Ada units, the RCI assigns a unique default remote name to each host Ada unit, but the remote name can be changed. For secondary referencers, the default remote name is the same as the name of the secondary text file on the host. You must make sure that the remote name is unique. Use the Rci.Set_Remote_Unit_Name command to change remote names if necessary.

Parameters

`Unit : String := "<CURSOR>";`

Specifies a naming expression for one or more units in an RCI view whose remote names are to be displayed. If Unit is an Ada unit, the remote name of its associated remote unit is displayed. If Unit is a secondary referencer, the name of the remote secondary text file is displayed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Restrictions

The Unit parameter must specify either an Ada unit or a secondary referencer. If not, a warning message is issued and the Unit parameter is ignored.

References

"Remote Files and Names," page 56
procedure Set_Remote_Unit_Name

PROCEDURE SHOW_SECONDARY

```
procedure Show_Secondary
         (Primary_Unit  : String  := "<CURSOR>";
          Response      : String  := "<PROFILE>");
```

Description

Displays information about the secondaries associated with one or more primary Ada units.

The following information is displayed:

- Whether the primary is processed, based on the value of the Process_Primary flag
- Names of the secondary text units associated with the primary
- Remote names of the secondary text files
- Remote commands associated with the secondary text files

The Primary_Unit parameter can be a naming expression that refers to more than one unit in a view, so the display includes all of the named units (which must be Ada units).

Parameters

`Primary_Unit : String := "<CURSOR>";`

Specifies a naming expression for one or more Ada units in an RCI view whose secondaries and associated information should be displayed.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Restrictions

The specified primary unit must be an Ada unit.

References

Chapter 7, "Using Non-Ada Code with the RCI"
procedure Create_Secondary
procedure Remove_Secondary

PROCEDURE SHOW_UNITS

```
procedure Show_Units
         (Unit        : String  := "<CURSOR>";
          Remote_Name : Boolean := True;
          Consistency : Boolean := False;
          Response    : String  := "<PROFILE>");
```

Description

Displays relevant information about the state or configuration of units, when given a naming expression describing some number of units in a single RCI view.

Parameters

`Unit : String := "<CURSOR>";`

Specifies a naming expression for one or more units in an RCI view.

`Remote_Name : Boolean := True;`

Specifies, if True, that the remote names of the units are displayed. If a unit is a secondary referencer, the displayed remote name is that of its associated secondary text file.

`Consistency : Boolean := False;`

Specifies, if True, that the remote units are checked for changes since they were last downloaded.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

Restrictions

If the unit is not an Ada unit or a secondary referencer, a warning message is displayed and the Unit parameter is ignored.

PROCEDURE TRANSFER_UNITS

```
procedure Transfer_Units
         (Units          : String  := "<CURSOR>";
          Remote_Machine : String  := "";
          Effort_Only    : Boolean := False;
          Response       : String  := "<PROFILE>");
```

Description

Transfers all Ada units and any secondaries specified by the Units parameter to the specified remote machine, if the host and remote machine can communicate through FTP.

Often there is a need to do multimachine development where a host view maps onto many target libraries. Under these conditions, it is useful to generate a batch script and a list of units to build once and then run this batch script on each of the multiple machines. To do this:

1. Use the Build_Script or the Build_Script_Via_Tape command to generate a host script file and a file containing a list of the units to build (a Build_List_File).
2. For each remote machine:
   a. Download the units that are to be built using the Transfer_Units command.
   b. Download and execute the remote script file using the Execute_Script command.

Parameters

`Units : String := "<CURSOR>";`

Specifies a naming expression describing the host units that are to be downloaded to the specified remote machine. Units can also specify an indirect file that contains the names of the host units to download; for example, to download the units selected for the last build, you can supply the name of the Build_List_File generated by Rci.Build_Script.

`Remote_Machine : String := "";`

Specifies the name of the remote machine onto which to download the specified units.

`Effort_Only : Boolean := False;`

Specifies, if True, that the names of the files that would be downloaded are displayed, but none of them are actually moved to the remote machine.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Downloading Host Units," page 71
procedure Build_Script
procedure Build_Script_Via_Tape
procedure Execute_Script

PROCEDURE UPLOAD_ASSOCIATED_FILES

```
procedure Upload_Associated_Files
         (Units       : String  := "<CURSOR>";
          Effort_Only : Boolean := False;
          Response    : String  := "<PROFILE>");
```

Description

Uploads the associated files for the specified units, if the host and remote machine can communicate through FTP.

This call may generate errors if a batch script was generated but never run on the remote compilation platform for the specified units.

Parameters

`Units : String := "<CURSOR>";`

Specifies a naming expression describing the host units for which to upload associated files. Units can also specify an indirect file that contains the names of the host units whose associated files are to be uploaded. For example, to load the associated files of units selected for the last build, you can supply the name of the Build_List-_File generated by Rci.Build_Script.

`Effort_Only : Boolean := False;`

Specifies, if True, that the names of the files that would be uploaded are displayed, but none of them are actually moved to the host machine.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Retrieving Associated Files," page 72

PROCEDURE UPLOAD_UNIT

```
procedure Upload_Unit
          (Remote_Unit_Name     : String   := ">>REMOTE UNIT NAME<<";
           Into_View            : String   := "<CURSOR>";
           Upload_To_Text_File  : Boolean  := False;
           Host_Text_File_Name  : String   := "";
           Response             : String   := "<PROFILE>");
```

Description

Transfers a unit from the remote machine into a host view.

This command should be used on objects that do not currently reside in the host view. This allows you to move remotely maintained/created code to a Rational view. This command should be used to upload only new units, which were created on the remote machine and do not exist on the host, into the host view.

If Upload_To_Text_File is False, the unit is parsed into an Ada unit, and the host unit name will be the name of the compilation unit. In this case, if there is an existing unit with the same name, it must be in source state to be replaced by the new unit.

If Upload_To_Text_File is True, no Ada parsing is done and the remote unit is uploaded into Host_Text_File_Name. If a text file of the same name already exists, it is overwritten with the new file.

Parameters

`Remote_Unit_Name : String := ">>REMOTE UNIT NAME<<";`

Specifies the name of the unit in the remote directory that is to be uploaded.

`Into_View : String := "<CURSOR>";`

Specifies an RCI view.

`Upload_To_Text_File : Boolean := False;`

Specifies, if True, that no attempt is made to parse the remote object into an Ada unit. In this case, a Host_Text_File_Name must be supplied.

`Host_Text_File_Name : String := "";`

Specifies, for uploading a non-Ada unit, the name of the host text file into which the remote text should be uploaded. This must be a valid, unambiguous naming expression. If it is a simple name, the unit is uploaded into Into_View. If it is a full name, the unit is uploaded into the specified filename, even if that file is not contained in Into_View.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Determining Consistency of Host and Remote Units," page 78
"Uploading a New Remote Unit to the Host," page 80
"Changing Text from the Remote Machine," page 103

PROCEDURE UPLOAD_UNITS

```
procedure Upload_Units
         (Upload_Specification_File  : String := "";
          Into_View                 : String := "<CURSOR>";
          Response                  : String := "<PROFILE>");
```

Description

Uploads remote units into host units as specified by the Upload_Specification_File parameter.

Use this command only to upload new units, which were created on the remote machine and do not exist on the host, into the host view directory.

Upload_Specification_File contains the list of units to be uploaded. Each line of the file has two fields separated by blanks:

- Remote-unit name: The name of a unit on the remote system.
- Host text filename: The new host name for the uploaded unit. If this field is left blank, the command attempts to upload the remote unit into an Ada name in Into_View on the host. The RCI uses the naming scheme for remote units that has been defined by your extension to create the Ada unit name on the host.

A sample file appears as follows:

```
x.c          x_c
new_unit.c       unit_in_c
y_s.ada
```

Remote file x.c is copied to host text file X_C in Into_View; remote unit new_unit.c is copied to Unit_In_C; remote file y_s.ada is copied to host Ada unit Y'Spec.

Parameters

`Upload_Specification_File : String := "";`

Specifies the file, described above, that contains information on the units to upload.

`Into_View : String := "<CURSOR>";`

Specifies an RCI view.

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

Specifies how to respond to errors, where to send log messages, and what activity to use during the execution of this command. By default, this command uses the response characteristics specified in the job response profile for the current job. For other values accepted by this parameter, see Parameter-Value Conventions in the Reference Summary (RS) of the *Rational Environment Reference Manual.*

References

"Uploading a New Remote Unit to the Host," page 80

9

Package Rci_Cmvc

This chapter provides an overview of the functionality of the commands in package Rci_Cmvc. Details on how to use and apply these commands are given in earlier chapters, and those chapters are referenced here.

The library-management operations described in this chapter are affected by the values of the following switches:

- Operation_Mode (to determine Batch or Interactive mode)
- Session_Rci.Auto_Create_Remote_Directory switch (for view creation)
- Rci.Host_Only switch (for existing views)

Operations may also vary depending on values specific to your extension. See your extension user's guide or your customizer for more information.

Following the overview, all of the commands in package Rci_Cmvc are listed in alphabetical order with a description of how their operation differs from the Cmvc commands of the same name.

The commands in package Rci_Cmvc provide the same functionality as the corresponding Cmvc commands. The operations in this group are built on top of the CMVC operations of the same names. They expand the CMVC operations by allowing you to explicitly specify the Remote_Machine and Remote_Directory parameters for each command. This allows you to define the remote machine and directory rather than use the default constructed by library-switch settings in your environment. These commands are available for situations in which the default naming scheme does not support your naming system or the default names are too long.

RCI COMMANDS FOR CMVC

Package Rci_Cmvc contains the following commands for creating, releasing, destroying, and recreating views:

- Build: Rebuilds the host view from saved historical information.
- Copy: Copies a new view from an existing view and creates the remote library for that new view.
- Initial: Creates a new RCI combined view or a new subsystem and RCI combined view.
- Make_Path: Creates a new RCI combined view from an existing RCI or R1000 view.
- Make_Subpath: Creates a new RCI combined view from an existing RCI view.
- Release: Creates a release of an RCI view.
- Make_Spec_View: Creates a new RCI spec view from each of the specified RCI views in a spec/load subsystem.

REMOTE_MACHINE AND REMOTE_DIRECTORY PARAMETERS

Rci_Cmvc operations accept the two additional parameters:

- `Remote_Machine : String := "";`

  Specifies the machine where the remote library will be located. The value specified is assigned to the Ftp.Remote_Machine switch in the library-switch or session-switch file of the view that is created. If the default is used, units cannot be brought to the coded state in the combined view (and in the remote directory).

- `Remote_Directory : String := "";`

  Specifies, for view-creation commands, the full pathname of the library on the remote machine with which this view will be associated. If the library does not already exist, it is created. The value specified is assigned to the Ftp.Remote_Directory switch in the library-switch or session-switch file of the view being created. If the default is used, units cannot be brought to the coded state in the combined view (and in the target program library).

  For Rci_Cmvc.Release, this parameter specifies the name of a remote library that should be built to match the host released view. If the default null string is used, only the host component (view) is released. If a nonnull string is provided, the remote directory is also released.

To view the default values for Remote_Machine and Remote_Directory that will be used if these parameters are not specified, use the Rci.Display_Default_Naming command.

PROCEDURE BUILD

```
procedure Build
         (Configuration    : String  := ">>CONFIGURATION NAME<<";
          Remote_Machine   : String  := "";
          Remote_Directory : String  := "";
          View_To_Import   : String  := "<INHERIT_IMPORTS>";
          Model            : String  := "<INHERIT_MODEL>";
          Goal             : Compilation.Unit_State
                                     := Compilation.Installed;
          Limit            : String  := "<WORLDS>";
          Comments         : String  := "";
          Work_Order       : String  := "<DEFAULT>";
          Volume           : Natural := 0;
          Response         : String  := "<PROFILE>");
```

Description

Rebuilds the host view from saved historical information.

If the Configuration parameter refers to a text file, that file is assumed to be an indirect file containing a list of the names of configuration objects that are to be built.

The Unit_State for the Goal parameter is limited to Compilation.Installed, since only the host view is rebuilt.

Parameters

See Cmvc.Build and "Remote_Machine And Remote_Directory Parameters" on page 160.

References

"Creating New RCI Views," page 39
Chapter 6, "Library Management"
procedure Cmvc.Build, page 173
procedure Cmvc.Build, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE COPY

```
procedure Copy
   (From_View                : String  := "<CURSOR>";
    New_Working_View         : String  := ">>SUB/PATH NAME<<";
    Remote_Machine           : String  := "";
    Remote_Directory         : String  := "";
    View_To_Modify           : String  := "";
    View_To_Import           : String  := "<INHERIT_IMPORTS>";
    Only_Change_Imports      : Boolean := True;
    Join_Views               : Boolean := True;
    Reservation_Token_Name   : String  := "<AUTO_GENERATE>";
    Construct_Subpath_Name   : Boolean := False;
    Create_Spec_View         : Boolean := False;
    Create_Load_View         : Boolean := False;
    Create_Combined_View     : Boolean := False;
    Level_For_Spec_View      : Natural := 0;
    Model                    : String  := "<INHERIT_MODEL>";
    Remake_Demoted_Units     : Boolean := True;
    Goal                     : Compilation.Unit_State
                                        := Compilation.Coded;
    Comments                 : String  := "";
    Work_Order               : String  := "<DEFAULT>";
    Volume                   : Natural := 0;
    Response                 : String  := "<PROFILE>");
```

Description

Creates one or more new working views by copying the specified view or views.

Parameters

See Cmvc.Copy and "Remote_Machine And Remote_Directory Parameters" on page 160.

References

procedure Cmvc.Copy, page 174
procedure Cmvc.Copy, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE INITIAL

```
procedure Initial
         (Subsystem              : String  := "<IMAGE>";
          Working_View_Base_Name : String  := "Rev1";
          Remote_Machine         : String  := "";
          Remote_Directory       : String  := "";
          Subsystem_Type         : Cmvc.System_Object_Enum
                                       := Cmvc.Spec_Load_Subsystem;
          View_To_Import         : String  := "";
          Model                  : String  := ">>RCI MODEL<<";
          Comments               : String  := "";
          Work_Order             : String  := "<DEFAULT>";
          Volume                 : Natural := 0;
          Response               : String  := "<PROFILE>");
```

Description

Builds a new combined RCI view in an existing subsystem or builds a new subsystem and combined RCI view.

The operation of this command is identical to that described for Cmvc.Initial on page 175, with the exception that, to create a remote library, the Remote_Machine and Remote_Directory parameters must be nonblank.

Parameters

See Cmvc.Initial and "Remote_Machine And Remote_Directory Parameters" on page 160.

References

"Creating a Subsystem and an RCI View," page 36
Chapter 6, "Library Management"
procedure Cmvc.Initial, page 175
procedure Cmvc.Initial, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE MAKE_PATH

```
procedure Make_Path
        (From_Path              : String   := "<CURSOR>";
         New_Path_Name          : String   := ">>PATH NAME<<";
         Remote_Machine         : String   := "";
         Remote_Directory       : String   := "";
         View_To_Modify         : String   := "";
         View_To_Import         : String   := "<INHERIT_IMPORTS>";
         Only_Change_Imports    : Boolean  := True;
         Model                  : String   := "<INHERIT_MODEL>";
         Join_Paths             : Boolean  := True;
         Remake_Demoted_Units   : Boolean  := True;
         Goal                   : Compilation.Unit_State
                                       := Compilation.Installed;
         Comments               : String   := "";
         Work_Order             : String   := "<DEFAULT>";
         Volume                 : Natural  := 0;
         Response               : String   := "<PROFILE>");
```

Description

Builds a path that is a combined view from an existing R1000 or RCI view.

The operation of this command is identical to that described for Cmvc.Make_Path on page 176, with the following exceptions:

- Make_Path checks that Remote_Directory, if specified, is not used by any of the views within the subsystem of the new path. If this parameter is allowed to default to the null string, units in the new path can be promoted only to the installed state as described in "Creating New RCI Views," page 38 in Chapter 2.
- By default, units copied to the new path are promoted only to the installed state; they cannot be promoted to the coded state unless they can be downloaded and compiled on the remote machine. The Remote_Machine and Remote_Directory parameters must be set to create units on the remote machine.

Parameters

See Cmvc.Make_Path and "Remote_Machine And Remote_Directory Parameters" on page 158.

References

"Creating New RCI Views," page 38
Chapter 6, "Library Management"
procedure Cmvc.Make_Path, page 176
procedure Cmvc.Make_Path, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE MAKE_SPEC_VIEW

```
procedure Make_Spec_View
         (From_Path           : String  := "<CURSOR>";
          Spec_View_Prefix    : String  := ">>PREFIX<<";
          Level               : Natural := 0;
          Remote_Machine      : String  := "";
          Remote_Directory    : String  := "";
          View_To_Modify      : String  := "";
          View_To_Import      : String  := "<INHERIT_IMPORTS>";
          Only_Change_Imports : Boolean := True;
          Remake_Demoted_Unit : Boolean := True;
          Goal                : Compilation.Unit_State
                                          := Compilation.Coded;
          Comments            : String  := "";
          Work_Order          : String  := "<DEFAULT>";
          Volume              : Natural := 0;
          Response            : String  := "<PROFILE>");
```

Description

Builds a spec view from an existing RCI view.

For more information, see the Make_Path command in this section and Cmvc-.Make_Spec_View in the Project Management (PM) book of the *Rational Environment Reference Manual.*

Parameters

See Cmvc.Make_Spec_View and "Remote_Machine and Remote_Directory Parameters" on page 160.

References

"Creating New RCI Views," page 38
Chapter 6, "Library Management"
procedure Cmvc.Make_Spec_View, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE MAKE_SUBPATH

```
procedure Make_Subpath
        (From_Path              : String  := "<CURSOR>";
         New_Subpath_Extension  : String  := ">>SUBPATH<<";
         Remote_Machine         : String  := "";
         Remote_Directory       : String  := "";
         View_To_Modify         : String  := "";
         View_To_Import         : String  := "<INHERIT_IMPORTS>";
         Only_Change_Imports    : Boolean := True;
         Remake_Demoted_Units   : Boolean := True;
         Goal                   : Compilation.Unit_State
                                       := Compilation.Installed;
         Comments               : String  := "";
         Work_Order             : String  := "<DEFAULT>";
         Volume                 : Natural := 0;
         Response               : String  := "<PROFILE>");
```

Description

Builds a subpath that is a combined view from an existing RCI view.

The operation of this command is identical to the Cmvc.Make_Subpath command on page 176 with the addition of the Remote_Machine and Remote_Directory parameters, as described in the Rci_Cmvc.Make_Path command in this chapter.

Parameters

See Cmvc.Make_Subpath and "Remote_Machine and Remote_Directory Parameters" on page 160.

References

"Creating New RCI Views," page 38
Chapter 6, "Library Management"
procedure Cmvc.Make_Path, page 176
procedure Cmvc.Make_Subpath, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE RELEASE

```
procedure Release
    (From_Working_View          : String  := "<CURSOR>";
     Release_Name               : String  := "<AUTO_GENERATE>";
     Remote_Machine             : String  := "";
     Remote_Directory           : String  := "";
     Level                      : Natural := 0;
     Views_To_Import            : String  := "<INHERIT_IMPORTS>";
     Create_Configuration_Only  : Boolean := False;
     Compile_The_View           : Boolean := True;
     Goal                       : Compilation.Unit_State
                                       := Compilation.Installed;
     Comments                   : String  := "";
     Work_Order                 : String  := "<DEFAULT>";
     Volume                     : Natural := 0;
     Response                   : String  := "<PROFILE>");
```

Description

Creates a release of an RCI view.

This command provides the same functionality as the Cmvc.Release command described on page 180, with the exception of the addition of the Remote_Machine and Remote_Directory parameters.

Parameters

See Cmvc.Release and "Remote_Machine and Remote_Directory Parameters" on page 160.

References

"Creating Releases of Views," page 96
procedure Cmvc.Release, page 180
procedure Cmvc.Release, in the Project Management (PM) book of the *Rational Environment Reference Manual*

10 Package Cmvc

This chapter provides an overview of the extended RCI functionality of the commands in package Cmvc. Details on how to use and apply these commands are given in earlier chapters, and those chapters are referenced here. For more information about the commands, refer to the CMVC section of the Project Management (PM) book of the *Rational Environment Reference Manual.*

The library-management operations described in this chapter are affected by the values of the following switches:

- Operation_Mode (to determine Batch or Interactive mode)
- Session_Rci.Auto_Create_Remote_Directory switch (for view creation)
- Rci.Host_Only switch (for existing views)

Operations may also vary depending on values specific to your extension. See your extension user's guide or your customizer for more information.

Following the overview, all of the commands with extended RCI functionality in package Cmvc are listed in alphabetical order with a description of how their operation differs when developing in an RCI environment.

The commands in package Cmvc are divided into several logical groupings that reflect the association between the commands and the operations that you need to perform while developing in an RCI environment. These groupings are discussed in the following sections.

LIBRARY OPERATIONS

The CMVC operations in this group extend functionality for the RCI by manipulating RCI state information where necessary. For example, when a new view is created, the appropriate RCI state information is also created. These commands also manipulate the remote libraries associated with the host views.

- Initial: Creates a new RCI combined view or a new subsystem and RCI combined view.
- Make_Path: Creates a new RCI combined view from an existing RCI or R1000 view.
- Make_Subpath: Creates a new RCI combined view from an existing RCI view.
- Build: Rebuilds the host view from saved historical information.
- Copy: Copies a new view from an existing view.
- Destroy_View: Demotes all units in the RCI view and removes the view.
- Release: Creates a release of an RCI view.
- Make_Spec_View: Creates a new RCI spec view from each of the specified RCI views in a spec/load subsystem.

IMPORT OPERATIONS

The following CMVC commands have been extended to support host and remote imports for the RCI:

- Import: Adds an imported view to a view's import list.
- Remove_Import: Removes an imported view from a view's import list.

CONSISTENCY-MANAGEMENT OPERATIONS

CMVC ensures that only one user at a time can update a unit joined across many views on the host. However, remote code that is associated with code in RCI views can be updated at will. It is the responsibility of the user to make sure that host and remote source remain consistent. The following operations provide support for management of consistency between host and remote objects:

- Accept_Changes: Accepts changes from another object.
- Revert: Replaces the contents of an object with the contents from a specified generation.
- Abandon_Reservation: Abandons the checkout reservation on an object.

PROCEDURE ABANDON_RESERVATION

```
procedure Abandon_Reservation
         (What_Object           : String  := "<SELECTION>";
          Allow_Demotion        : Boolean := False;
          Remake_Demoted_Units  : Boolean := True;
          Goal                  : Compilation.Unit_State
                                        := Compilation.Coded;
          Comments              : String  := "";
          Work_Order            : String  := "<DEFAULT>";
          Response              : String  := "<PROFILE>");
```

Description

Abandons a checkout of some object or set of objects.

For the RCI, if the file that is being abandoned is a secondary text file, then the primary associated with the file is demoted. The secondary text is unfrozen.

Parameters

See Cmvc.Abandon_Reservation.

References

procedure Cmvc.Abandon_Reservation, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE ACCEPT_CHANGES

```
procedure Accept_Changes
        (Destination          : String  := "<CURSOR>";
         Source               : String  := "<LATEST>";
         Allow_Demotion       : Boolean := False;
         Remake_Demoted_Units : Boolean := True;
         Goal                 : Compilation.Unit_State
                              := Compilation.Coded;
         Comments             : String  := "";
         Work_Order           : String  := "<DEFAULT>";
         Response             : String  := "<PROFILE>");
```

Description

Accepts changes from one object to another object.

The following special actions are taken for RCI secondary text files and secondary-referencer files:

- If changes are being accepted into a secondary text file, then the primary unit associated with the secondary text file is demoted. Specifying Remake_Demoted_Units as True causes the demoted units to be promoted.
- Using Accept_Changes on a primary unit specifying both source and destination views causes the secondary referencers of the source primary to be copied to the secondary referencers of the destination primary.
- Using Accept_Changes on a primary unit specifying <LATEST> as the source does not copy any secondary referencers.
- Using Accept_Changes unfreezes all text files in the destination view.

Parameters

See Cmvc.Accept_Changes.

References

procedure Cmvc.Accept_Changes, in the Project Management (PM) book of the *Rational Environment Reference Manual*)

PROCEDURE BUILD

```
procedure Build
        (Configuration  : String  := ">>CONFIGURATION NAME<<";
         View_To_Import : String  := "<INHERIT_IMPORTS>";
         Model          : String  := "<INHERIT_MODEL>";
         Goal           : Compilation.Unit_State
                                  := Compilation.Installed;
         Limit          : String  := "<WORLDS>";
         Comments       : String  := "";
         Work_Order     : String  := "<DEFAULT>";
         Volume         : Natural := 0;
         Response       : String  := "<PROFILE>");
```

Description

Rebuilds the host view from saved historical information.

If the Configuration parameter refers to a text file, that file is assumed to be an indirect file that contains a list of the names of configuration objects that are to be built.

The Unit_State for the Goal parameter is limited to Compilation.Installed, since only the host view is rebuilt.

Parameters

See Cmvc.Build.

References

"Creating New RCI Views," page 38
Chapter 6, "Library Management"
procedure Cmvc.Build, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE COPY

```
procedure Copy
   (From_View                : String  := "<CURSOR>";
    New_Working_View         : String  := ">>SUB/PATH NAME<<";
    View_To_Modify           : String  := "";
    View_To_Import           : String  := "<INHERIT_IMPORTS>";
    Only_Change_Imports      : Boolean := True;
    Join_Views               : Boolean := True;
    Reservation_Token_Name   : String  := "<AUTO_GENERATE>";
    Construct_Subpath_Name   : Boolean := False;
    Create_Spec_View         : Boolean := False;
    Create_Load_View         : Boolean := False;
    Create_Combined_View     : Boolean := False;
    Level_For_Spec_View      : Natural := 0;
    Model                    : String  := "<INHERIT_MODEL>";
    Remake_Demoted_Units     : Boolean := True;
    Goal                     : Compilation.Unit_State
                                       := Compilation.Coded;
    Comments                 : String  := "";
    Work_Order               : String  := "<DEFAULT>";
    Volume                   : Natural := 0;
    Response                 : String  := "<PROFILE>");
```

Description

Creates one or more new working views by copying the specified view or views.

Parameters

See Cmvc.Copy.

References

procedure Cmvc.Copy, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE DESTROY_VIEW

```
procedure Destroy_View
         (What_View                    : String  := "<SELECTION>";
          Demote_Clients               : Boolean := False;
          Destroy_Configuration_Also   : Boolean := False;
          Comments                     : String  := "";
          Work_Order                   : String  := "<DEFAULT>";
          Response                     : String  := "<PROFILE>");
```

Description

Destroys a host view and its corresponding remote library.

If you want to destroy the host view but not the associated remote directory, set the view's Rci.Host_Only switch to True.

Cmvc.Destroy_View demotes all units before performing the destroy operation. If the RCI state information has already been destroyed, preventing normal demotion, this routine forces the demotion of units in the indicated view before performing the destroy operation. Use the Rci.Refresh_View command to rebuild the state information and then reissue the Destroy_View command.

The configuration object for the view is left in its normal place so the view can be reconstructed using the Build command.

Parameters

See Cmvc.Destroy_View.

References

"Removing RCI Views," page 95
procedure Cmvc.Destroy_View, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE IMPORT

```
procedure Import
         (View_To_Import       : String   := "<REGION>";
          Into_View            : String   := "<CURSOR>";
          Only_Change_Imports  : Boolean  := False;
          Import_Closure       : Boolean  := False;
          Remake_Demoted_Units : Boolean  := True;
          Goal                 : Compilation.Unit_State
                                          := Compilation.Coded;
          Comments             : String   := "";
          Work_Order           : String   := "<DEFAULT>";
          Response             : String   := "<PROFILE>");
```

Description

Adds an imported view to a view's import list.

This operation may apply to remote program-library state, depending on the specific RCI customization.

Parameters

See Cmvc.Import.

References

"Imports," page 92
procedure Cmvc.Import, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE INITIAL

```
procedure Initial
  (System_Object          : String   := ">>SYSTEM OBJECT NAME<<";
   Working_View_Base_Name : String   := "Rev1";
   System_Object_Type     : System_Object_Enum
                                     := Cmvc.Spec_Load_Subsystem;
   View_To_Import         : String   := "";
   Create_Load_View       : Boolean  := True;
   Model                  : String   := "R1000";
   Comments               : String   := "";
   Work_Order             : String   := "<DEFAULT>";
   Volume                 : Natural  := 0;
   Response               : String   := "<PROFILE>");
```

Description

Builds a new combined RCI view in an existing subsystem or builds a new subsystem and combined RCI view.

In addition to creating a new view, this routine initializes the view's RCI state, as described in "Rci State Information" on page 94 in Chapter 6.

If the following items are true, a remote library is also constructed:

- Remote extensions management is enabled, as described in Chapter 2, "RCI Setup Operations"
- The remote username and password are set in the user's session switches or in the Remote_Passwords file, as described in "Specifying Remote Login Information" on page 20 in Chapter 2.
- The default switch-naming scheme provides values for Remote_Machine and Remote_Directory.

In Interactive mode, if any of the above is not true or if the remote library cannot be built for any other reason, it is not possible to bring units to the coded state in the combined view until the remote library is built explicitly by the user (as described in Chapter 6) and the other conditions are also corrected.

Parameters

See Cmvc.Initial.

References

"Creating a Subsystem and an RCI View," page 36
Chapter 6, "Library Management"
procedure Cmvc.Initial, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE MAKE_PATH

```
procedure Make_Path
        (From_Path              : String  := "<CURSOR>";
         New_Path_Name          : String  := ">>PATH NAME<<";
         View_To_Modify         : String  := "";
         View_To_Import         : String  := "<INHERIT_IMPORTS>";
         Only_Change_Imports    : Boolean := True;
         Create_Load_View       : Boolean := False;
         Create_Combined_View   : Boolean := False;
         Model                  : String  := "<INHERIT_MODEL>";
         Join_Paths             : Boolean := True;
         Remake_Demoted_Units   : Boolean := True;
         Goal                   : Compilation.Unit_State
                                        := Compilation.Coded;
         Comments               : String  := "";
         Work_Order             : String  := "<DEFAULT>";
         Volume                 : Natural := 0;
         Response               : String  := "<PROFILE>");
```

Description

Builds a path that is a combined view from an existing R1000 or RCI view.

In addition to building a new path, this command initializes the RCI state for the new view as described in "Rci State Information" on page 94 in Chapter 6.

When a new path is built from an existing view, a new remote directory must be used to prevent two views in the same subsystem from overwriting the same remote library. Make_Path checks that the Remote_Directory value specified by default switches is not used by any of the views within the subsystem of the new path. If this parameter is allowed to default to the null string, units in the new path can be promoted to the installed state only as described in "Creating New RCI Views" on page 38 in Chapter 2.

If the specified remote library does not exist, or exists on the specified remote machine and is not referenced by any other view in the same subsystem, that library is associated with the new view.

If the new path is being built from an R1000 view, a new model world and new imports appropriate to the RCI must be specified.

Imports must be from other combined views with the same target key as the new path.

By default, Make_Path attempts to promote units copied to the new path to the coded state; to do this, Make_Path must be able to download and compile the units on the remote machine.

Secondary referencers and secondary text files that are copied are not frozen in the new view.

Parameters

See Cmvc.Make_Path.

References

"Creating New RCI Views," page 38
Chapter 6, "Library Management"
procedure Cmvc.Make_Path, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE MAKE_SPEC_VIEW

```
procedure Make_Spec_View
  (From_Path            : String  := "<CURSOR>";
   Spec_View_Prefix     : String  := ">>PREFIX<<";
   Level                : Natura  := 0;
   View_To_Modify       : String  := "";
   View_To_Import       : String  := "<INHERIT_IMPORTS>";
   Only_Change_Imports  : Boolean := True;
   Remake_Demoted_Units : Boolean := True;
   Goal                 : Compilation.Unit_State
                                  := Compilation.Coded;
   Comments             : String  := "";
   Work_Order           : String  := "<DEFAULT>";
   Volume               : Natura  := 0;
   Response             : String  := "<PROFILE>");
```

Description

Makes a spec view for a path.

Parameters

See Cmvc.Make_Spec_View.

References

procedure Cmvc.Make_Spec_View, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE MAKE_SUBPATH

```
procedure Make_Subpath
        (From_Path              : String  := "<CURSOR>";
         New_Subpath_Extension  : String  := ">>SUBPATH<<";
         View_To_Modify         : String  := "";
         View_To_Import         : String  := "<INHERIT_IMPORTS>";
         Only_Change_Imports    : Boolean := True;
         Remake_Demoted_Units   : Boolean := True;
         Goal                   : Compilation.Unit_State
                                        := Compilation.Coded;
         Comments               : String  := "";
         Work_Order             : String  := "<DEFAULT>";
         Volume                 : Natural := 0;
         Response               : String  := "<PROFILE>");
```

Description

Builds a subpath that is a combined view from an existing RCI view.

Parameters

See Cmvc.Make_Subpath.

References

"Creating New RCI Views," page 38
Chapter 6, "Library Management"
procedure Make_Path in this chapter
procedure Cmvc.Make_Subpath, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE RELEASE

```
procedure Release
  (From_Working_View          : String   := "<CURSOR>";
   Release_Name               : String   := "<AUTO_GENERATE>";
   Level                      : Natural  := 0;
   Views_To_Import            : String   := "<INHERIT_IMPORTS>";
   Create_Configuration_Only  : Boolean  := False;
   Compile_The_View           : Boolean  := True;
   Goal                       : Compilation.Unit_State
                              := Compilation.Coded;
   Comments                   : String   := "";
   Work_Order                 : String   := "<DEFAULT>";
   Volume                     : Natural  := 0;
   Response                   : String   := "<PROFILE>");
```

Description

Creates a release of an RCI view.

When operating on an RCI view, in addition to the standard release functions, this command includes the following actions:

- Releases the associated remote library
- Releases secondary referencers

Parameters

See Cmvc.Release.

References

"Creating Releases of Views," page 96
procedure Cmvc.Release, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE REMOVE_IMPORT

```
procedure Remove_Import
         (View        : String  := ">>VIEW NAME<<";
          From_View   : String  := "<CURSOR>";
          Comments    : String  := "";
          Work_Order  : String  := "<DEFAULT>";
          Response    : String  := "<PROFILE>");
```

Description

Removes an imported view from a view's import list.

This command removes an import from the host view and from the view's associated remote library.

This operation may apply to remote program-library state, depending on the specific RCI extention.

Parameters

See Cmvc.Remove_Import.

References

"Imports," page 92
procedure Cmvc.Remove_Import, in the Project Management (PM) book of the *Rational Environment Reference Manual*

PROCEDURE REVERT

```
procedure Revert
         (What_Object             : String  := "<SELECTION>";
          To_Generation           : Integer := -1;
          Make_Latest_Generation  : Boolean := False;
          Allow_Demotion          : Boolean := False;
          Remake_Demoted_Units    : Boolean := True;
          Goal                    : Compilation.Unit_State
                                          := Compilation.Coded;
          Comments                : String  := "";
          Work_Order              : String  := "<DEFAULT>";
          Response                : String  := "<PROFILE>");
```

Description

Replaces the contents of the specified objects with the contents of the specified generation.

For the RCI, if the file that is being reverted is a secondary text file, then the primary associated with the file is demoted.

Parameters

See Cmvc.Revert.

References

procedure Cmvc.Revert, in the Project Management (PM) book of the *Rational Environment Reference Manual*

A Location of Components

This appendix lists the locations of important pieces of the RCI.

RCI software of interest to the user is delivered in two top-level libraries:

- !Model
- !Targets
  - !Targets.*Custom_Key*
  - !Targets.Implementation

!MODEL LIBRARY

The contents of the !Model library are discussed in Chapter 2, "RCI Setup Operations." The model worlds in the !Model library contain links to the !Targets library. Model worlds for the chosen target generally are located in:

`!Model.`*`Custom_Key`*

!TARGETS LIBRARY

The !Targets library consists of the following sublibraries:

- *Custom_Key*: Contains predefined packages and utilities, including those required by the Ada LRM and those required to interface to the target operating system.
- Implementation: Contains the subsystems that implement the RCI on the R1000.

Predefined Packages and Utilities (!Targets.*Custom_Key*)

Predefined packages contain the Ada code that user programs may require to be linked with their application code.

- I/O packages: The predefined I/O packages Direct_Io, Sequential_Io, Text_Io, and Io_Exceptions can be found in directory:

  `!Targets.`*`Custom_Key`*`.Io`

- Ada LRM packages: The predefined packages System, Calendar, Unchecked-_Conversion, and Unchecked_Deallocation can be found in directory:

  `!Targets.`*`Custom_Key`*`.Lrm`

- Target-specific tools: These sophisticated RCI features provide advanced functionality for application programs written to run on the chosen target. These target-specific components can be found in directory:

  `!Targets.`*`Custom_Key`*`.Target_Interface`

RCI Components (!Targets.Implementation)

- RCI compiler (Start_Rci_Main):

  `!Targets.Implementation.Rci_User_Interface`

- Package Rci and package Rci_Cmvc:

  `!Targets.Implementation.Rci_User_Interface`

- Customization interface:

  `!Targets.Implementation.Rci_Customization_Interface`

- Customizations (includes customization templates and library and compiler extensions):

  `!Targets.Implementation.Rci_Customization.`*`Custom_Key`*

B

Command Summary

PACKAGE RCI

```
procedure Accept_Remote_Changes
   (Unit                        : String    := "<CURSOR>";
    Allow_Demotion              : Boolean   := False;
    Compare_Objects             : Boolean   := False;
    Remake_Demoted_Units        : Boolean   := True;
    Goal                        : Compilation.Unit_State
                                            := Compilation.Coded;
    Response                    : String    := "<PROFILE>");

procedure Build_Remote_Library
   (View                        : String    := "<CURSOR>";
    Remote_Machine              : String    := "<DEFAULT>";
    Remote_Directory            : String    := "<DEFAULT>";
    Response                    : String    := "<PROFILE>");

procedure Build_Script
   (Host_Units                  : String    := <IMAGE>";
    Link_Main_Units             : Boolean   := True;
    Transfer_To_Target          : Boolean   := True;
    Host_Script_File            : String    := "<DEFAULT>";
    Remote_Script_File          : String    := ">> FULL REMOTE NAME <<";
    Build_List_File             : String    := "<DEFAULT>";
    Execute_Script              : Boolean   := False;
    Effort_Only                 : Boolean   := False;
    Minimal_Recompilation       : Boolean   := True;
    Make_Units                  : Boolean   := False;
    Response                    : String    := "<PROFILE>");

procedure Build_Script_Via_Tape
   (Host_Units                  : String    := "<IMAGE>";
    Link_Main_Units             : Boolean   := True;
    Host_Script_File            : String    := "<DEFAULT>";
    Remote_Script_File          : String    := ">> FULL REMOTE NAME <<";
    Build_List_File             : String    := "<DEFAULT>";
    Format                      : String    := "R1000";
    Volume                      : String    := "";
    Label                       : String    := "rci_build";
    Logical_Device              : String    := "rci";
    Effort_Only                 : Boolean   := False;
    Minimal_Recompilation       : Boolean   := True;
    Make_Units                  : Boolean   := False;
    Response                    : String    := "<PROFILE>");
```

```
procedure Check_Consistency
   (Unit                        : String    := "<CURSOR>";
    Compare_Objects             : Boolean   := False;
    Response                    : String    := "<PROFILE>");

procedure Collapse_Secondary_Referencers
   (Directory                   : String    := "<CURSOR>";
    Response                    : String    := "<PROFILE>");

procedure Create_Secondary
   (Primary_Unit                : String    := "<CURSOR>";
    Command                     : String    := "";
    Secondary_Text              : String    := "";
    Remote_Name                 : String    := "<DEFAULT>";
    Process_Primary             : Boolean   := False;
    Response                    : String    := "<PROFILE>");

procedure Destroy_Remote_Library
   (View                        : String    := "<CURSOR>";
    Response                    : String    := "<PROFILE>");

procedure Display_Default_Naming
   (Potential_View              : String    := "";
    Target_Key                  : String    := "";
    Response                    : String    := "<PROFILE>");

procedure Display_Unit_Options
   (Units                       : String    := "<CURSOR>";
    Response                    : String    := "<PROFILE>");

procedure Edit_Secondary
   (Secondary_Referencer        : String    := "<CURSOR>";
    In_Place                    : Boolean   := False;
    Visible                     : Boolean   := True;
    Response                    : String    := "<PROFILE>");

procedure Execute_Remote_Command
   (Remote_Command              : String    := "";
    Remote_Machine              : String    := "<DEFAULT>";
    Remote_Username             : String    := "<DEFAULT>";
    Remote_Password             : String    := "<DEFAULT>";
    Remote_Directory            : String    := "<DEFAULT>";
    The_Key                     : String    := "<DEFAULT>";
    Response                    : String    := "<PROFILE>");

procedure Execute_Script
   (Host_Script_File            : String    := "<IMAGE>";
    Remote_Script_File          : String    := ">> FULL REMOTE NAME <<";
    Remote_Machine              : String    := "<DEFAULT>";
    Remote_Username             : String    := "<DEFAULT>";
    Remote_Password             : String    := "<DEFAULT>";
    Remote_Directory            : String    := "";
    Effort_Only                 : Boolean   := False;
    Display_Remote_Commands     : Boolean   := False;
    The_Key                     : String    := "<DEFAULT>";
    Response                    : String    := "<PROFILE>");
```

```
procedure Expand_Secondary_Referencers
   (Directory                  : String   := "<CURSOR>";
    Response                   : String   := "<PROFILE>");

procedure Link
   (Main_Unit                  : String   := "<CURSOR>";
    Make_Uncoded_Units         : Boolean  := False;
    Executable_Name            : String   := "";
    Response                   : String   := "<PROFILE>");

procedure Rebuild_Remote_Library
   (View                       : String   := "<CURSOR>";
    Remake_Demoted_Units       : Boolean  := True;
    Remote_Machine             : String   := "<DEFAULT>";
    Remote_Directory           : String   := "<DEFAULT>";
    Response                   : String   := "<PROFILE>");

procedure Refresh_Remote_Imports
   (View                       : String   := "<CURSOR>";
    Response                   : String   := "<PROFILE>");

procedure Refresh_View
   (View                       : String   := "<CURSOR>";
    Retain_History             : Boolean  := True;
    Response                   : String   := "<PROFILE>");

procedure Remove_Secondary
   (Secondary_Referencer       : String   := "<CURSOR>";
    Response                   : String   := "<PROFILE>");

procedure Remove_Unit_Option
   (Option_Switch              : String   := "<ALL>";
    Units                      : String   := "<CURSOR>";
    Response                   : String   := "<PROFILE>");

procedure Set_Process_Primary
   (Primary_Unit               : String   := "<CURSOR>";
    Value                      : Boolean  := False;
    Response                   : String   := "<PROFILE>");

procedure Set_Remote_Unit_Name
   (Remote_Name                : String   := ">>REMOTE NAME<<";
    Unit                       : String   := "<CURSOR>";
    Allow_Demotion             : Boolean  := True;
    Response                   : String   := "<PROFILE>");

procedure Set_Secondary_Command
   (Command                    : String   := "";
    Secondary_Referencer       : String   := "<CURSOR>";
    Response                   : String   := "<PROFILE>");

procedure Set_Unit_Option
   (Option_Switch              : String   := ">>OPTION<<";
    Switch_Value               : String;
    Units                      : String   := "<CURSOR>";
    Response                   : String   := "<PROFILE>");
```

```
procedure Set_Unit_Option
   (Option_Switch              : String := ">>OPTION<<";
    Switch_Value               : Boolean;
    Units                      : String := "<CURSOR>";
    Response                   : String := "<PROFILE>");

procedure Show_Build_State
   (Host_Units                 : String := "<CURSOR>";
    Execution_Closure          : Boolean:= False;
    Obsolete_Units_Only        : Boolean:= False;
    Response                   : String := "<PROFILE>");

procedure Show_Remote_Information
   (View                       : String := "<CURSOR>";
    Response                   : String := "<PROFILE>");

procedure Show_Remote_Unit_Name
   (Unit                       : String := "<CURSOR>";
    Response                   : String := "<PROFILE>");

procedure Show_Secondary
   (Primary_Unit               : String := "<CURSOR>";
    Response                   : String := "<PROFILE>");

procedure Show_Units
   (Unit                       : String := "<CURSOR>";
    Remote_Name                : Boolean:= True;
    Consistency                : Boolean:= False;
    Response                   : String := "<PROFILE>");

procedure Transfer_Units
   (Units                      : String  := "<CURSOR>";
    Remote_Machine             : String  := "";
    Effort_Only                : Boolean := False;
    Response                   : String  := "<PROFILE>");

procedure Upload_Associated_Files
   (Units                      : String  := "<CURSOR>";
    Effort_Only                : Boolean := False;
    Response                   : String  := "<PROFILE>");

procedure Upload_Unit
   (Remote_Unit_Name           : String  := ">>REMOTE UNIT NAME<<";
    Into_View                  : String  := "<CURSOR>";
    Upload_To_Text_File        : Boolean := False;
    Host_Text_File_Name        : String  := "";
    Response                   : String  := "<PROFILE>");

procedure Upload_Units
   (Upload_Specification_File : String  := "";
    Into_View                  : String  := "<CURSOR>";
    Response                   : String  := "<PROFILE>");
```

PACKAGE RCI_CMVC

```
procedure Build
   (Configuration              : String    := ">>CONFIGURATION
    NAME<<";
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    View_To_Import             : String    := "<INHERIT_IMPORTS>";
    Model                      : String    := "<INHERIT_MODEL>";
    Goal                       : Compilation.Unit_State
                                           := CompilationInstalled;
    Installed; Limit           : String    := "<WORLDS>";
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>")
    renames Cmvc.Build;

procedure Copy
   (From_View                  : String    := "<CURSOR>";
    New_Working_View           : String    := ">>SUB/PATH NAME<<";
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    View_To_Modify             : String    := "";
    View_To_Import             : String    := "<INHERIT_IMPORTS>";
    Only_Change_Imports        : Boolean   := True;
    Join_Views                 : Boolean   := True;
    Reservation_Token_Name     : String    := "<AUTO_GENERATE>";
    Construct_Subpath_Name     : Boolean   := False;
    Create_Spec_View           : Boolean   := False;
    Create_Load_View           : Boolean   := False;
    Create_Combined_View       : Boolean   := False;
    Level_For_Spec_View        : Natural   := 0;
    Model                      : String    := "<INHERIT_MODEL>";
    Remake_Demoted_Units       : Boolean   := True;
    Goal                       : Compilation.Unit_State
                                           := Compilation.Coded;
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>");

procedure Initial
   (Subsystem                  : String    := "<IMAGE>";
    Working_View_Base_Name     : String    := "Rev1";
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    Subsystem_Type             : Cmvc.System_Object_Enum
                                           := Cmvc.Spec_Load_Sub-
    system;
    View_To_Import             : String    := "";
    Model                      : String    := ">>RCI MODEL<<";
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>");
```

```
procedure Make_Path
   (From_Path                  : String    := "<CURSOR>";
    New_Path_Name              : String    := ">>PATH NAME<<";
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    View_To_Modify             : String    := "";
    View_To_Import             : String    := "<INHERIT_IMPORTS>";
    Only_Change_Imports        : Boolean   := True;
    Model                      : String    := "<INHERIT_MODEL>";
    Join_Paths                 : Boolean   := True;
    Remake_Demoted_Units       : Boolean   := True;
    Goal                       : Compilation.Unit_State
                                           := Compilation.Installed;
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>");

procedure Make_Spec_View
   (From_Path                  : String    := "<CURSOR>";
    Spec_View_Prefix           : String    := ">>PREFIX<<";
    Level                      : Natural   := 0;
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    View_To_Modify             : String    := "";
    View_To_Import             : String    := "<INHERIT_IMPORTS>";
    Only_Change_Imports        : Boolean   := True;
    Remake_Demoted_Units       : Boolean   := True;
    Goal                       : Compilation.Unit_State
                                           := Compilation.Coded;
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>");

procedure Make_Subpath
   (From_Path                  : String    := "<CURSOR>";
    New_Subpath_Extension      : String    := ">>SUBPATH<<";
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    View_To_Modify             : String    := "";
    View_To_Import             : String    := "<INHERIT_IMPORTS>";
    Only_Change_Imports        : Boolean   := True;
    Remake_Demoted_Units       : Boolean   := True;
    Goal                       : Compilation.Unit_State
                                           := Compilation.Installed;
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>");
```

```
procedure Release
   (From_Working_View          : String    := "<CURSOR>";
    Release_Name               : String    := "<AUTO_GENERATE>";
    Remote_Machine             : String    := "";
    Remote_Directory           : String    := "";
    Level                      : Natural   := 0;
    Views_To_Import            : String    := "<INHERIT_IMPORTS>";
    Create_Configuration_Only  : Boolean   := False;
    Compile_The_View           : Boolean   := True;
    Goal                       : Compilation.Unit_State
                                           := Compilation.Installed;
    Comments                   : String    := "";
    Work_Order                 : String    := "<DEFAULT>";
    Volume                     : Natural   := 0;
    Response                   : String    := "<PROFILE>");
```

C

Extension Tables

This appendix provides blank tables and lists that can be filled in by the customizer of the RCI for the convenience of the RCI end user. These tables and lists describe customization values for this extension.

- Target-key name (*Custom_Key*):
- Target (execution) architecture:
- Remote (compilation) architecture:
- Remote operating system:
- Third-party Ada compiler:

This appendix describes the following items, with references to the sections in this manual:

- Remote program libraries and remote import structures, page 92
- Default target compiler and linker commands
- Remote filename length, page 56
- Host switches and target-compiler options, page 50
- Package Standard types, page 7
- Predefined libraries, page 7
- Representation-clause restrictions, page 54
- Attributes, page 7
- Pragmas, page 54
- Associated files and names, page 57
- Remote library management, Chapter 6
- Batch-compilation support, Chapter 4
- Network-communications mechanism, page 5
- Troubleshooting

REMOTE PROGRAM LIBRARIES AND IMPORT LISTS

The default name of the remote program library is:

The default name of the remote import list, if used, is:

REMOTE COMMAND NAMES

The default target-compiler command is:

The default remote-linker command is:

REMOTE FILENAME LENGTH

The remote operating system may place a limit on the maximum length for filenames downloaded to it from the Rational Environment. That limit is: _____ characters. This information is useful as described in "Remote Files and Names" on page 56 in Chapter 3.

HOST SWITCHES AND TARGET-COMPILER OPTIONS

Host library switches whose names begin with the extension-dependent string *Custom_Key* specify options for the target compiler and linker. These switches are discussed, and examples are given, in "Setting Session and Library Switches" on page 45.

For more detailed information about these target options, refer to the target-compiler user's manual.

Your customizer should provide the actual switch names and the compiler options that they affect. The following tables provide space to list the RCI-supported compiler and linker options, as well as the switches that set them:

- Table C-1 lists the RCI-supported options that apply to the target compiler.
- Table C-1 lists the RCI-supported options that apply to both the target compiler and the remote linker.
- Table C-1 lists the RCI-supported options that apply to the remote linker.
- Table C-4 lists the host *Custom_Key* switches and their values.

Table C-1 Supported Compiler Options

Compiler Option	RCI Switch

Table C-2 Supported Compiler/Linker Options

Compiler Option	RCI Switch

Table C-2 Supported Compiler/Linker Options (continued)

Compiler Option	RCI Switch

Table C-3 Supported Linker Options

Linker Option	RCI Switch

Table C-4 Host RCI Library Switches

Name	Type	Default	Mapping to Target-Compiler Options
*Custom_Key*_Option_Name	Boolean /String	False/ True/ String value	Description of the option
*Custom_Key*_Default_Machine	String	Null string	For information on the use of this switch, refer to the "Remote Machine Name," subsection in Chapter 2 of the *RCI User's Manual.*
*Custom_Key*_Default_Roof	String	Null string	For information on the use of this switch, refer to the "Remote Directory," subsection in Chapter 2 of the *RCI User's Manual.*

Table C-4 Host RCI Library Switches (continued)

Name	Type	Default	Mapping to Target-Compiler Options

PACKAGE STANDARD TYPES

The customizable RCI allows the customizer to define implementation-dependent types in package Standard. Refer to the target-compiler Appendix F for the *Reference Manual for the Ada Programming Language* (Ada LRM) for more information. The predefined types for the extension are the following:

-
-
- INTEGER
- FLOAT
-
-
-
- DURATION

PREDEFINED LIBRARIES

This RCI extension provides the following predefined libraries from the target-compiler Appendix F:

- In !Targets.*Custom_Key*.Io:
  - Direct_Io
  - Io_Exceptions
  - Sequential_Io
  - Text_Io
  -
- In !Targets.*Custom_Key*_Lrm:
  - System
  -
  -
  -
- In !Targets.*Custom_Key*.Reusable_Components:
  -
  -
  -
  -
  -
- In !Targets.*Custom_Key*.Target_Interface:
  -
  -
  -
  -
  -

REPRESENTATION CLAUSES

The customizable RCF allows semantic checking to be performed on representation clauses, as described in "Using Representation Clauses" on page 54 of Chapter 3. The following limitations have been placed on representation clauses during semantic checking:

- Address clauses:

- Record clauses:

- Size clauses:

- Small clauses:

- Storage_Size clauses:

- Other:

ATTRIBUTES

The customizable RCI allows semantic checking to be performed on implementation-defined attributes. Table C-5 lists the implementation-defined attributes defined for this extension.

Table C-5 Target Attributes Recognized by RCI

Attribute	Definition	Function

Table C-5 Target Attributes Recognized by RCI (continued)

Attribute	Definition	Function

PRAGMAS

This extension may provide support for implementation-specific and predefined pragmas and pragma Interface support for other languages.

Implementation-Dependent Pragmas

The target compiler may support implementation-specific pragmas not found in the R1000 compiler. The customizable RCI allows the customizer to define pragmas that are recognized by the host semanticizer as described in "Using Implementation-Dependent Pragmas" in Chapter 3. Table C-6 lists these pragmas.

Refer to the target-compiler Appendix F of the Ada LRM for more detailed information about these pragmas.

Table C-6 Target-Compiler Pragmas Recognized by the RCI

Pragma	Parameters	Function

Table C-6 Target-Compiler Pragmas Recognized by the RCI (continued)

Pragma	Parameters	Function

Predefined Pragmas

This extension may support the following predefined pragmas as described in the target-compiler Ada LRM.

- Controlled
- Elaborate
- Inline
- Interface
- List
- Memory_Size
- Optimize
- Pack
- Page
- Priority
- Shared
- Storage_Unit
- Suppress
- System_Name

Pragma Interface

Pragma Interface supports the following languages:

- C
- FORTRAN
- Assembly
- Pascal
-
-

ASSOCIATED FILES

Files that are created on the remote compilation platform during compilation and link processes can then be uploaded to the development host into specially named files that are subordinate to host files.

Associated files are described in "Host Associated Files" in Chapter 3. Associated filenames and the switches that affect them are specified by the RCI extension; Table C-7 provides space for that information for this extension.

Table C-7 Suffixes Identifying Associated Files on the Rational Environment

Suffix	Description of File and RCI Switch Setting

MANAGING REMOTE LIBRARIES

Depending on your customization, the RCI provides support for automated remote library management. Your customizer should provide information about remote library management for your extension.

This extension may support the following remote library-management operations:

- Creating host views and their associated remote libraries
- Destroying host views and remote libraries
- Controlling host and remote imports
- Releasing host views and remote libraries

This section describes library-management operations that take place when the view's operation mode is Interactive. The values of the Session_Rci.Auto_Create-_Remote_Directory and the Rci.Host_Only switches also affect these operations.

The following subsections describe how these operations are performed under your extension. For more information about RCI remote library management, refer to Chapter 6.

Creating Views and Remote Libraries

The extension may provide operations that automatically create Ada directories and libraries on the remote machine when you create views on the host. The following commands accomplish this, provided that the Session_Rci.Auto_Create_Remote_Directory switch has its default value (True):

- Cmvc.Initial or Rci_Cmvc.Initial: Creates a new host subsystem containing an R1000 or RCI view; also creates a new host view in an existing subsystem. In either case, this command creates a remote directory and a remote Ada program library associated with the new host view.
- Cmvc.Make_Path or Rci_Cmvc.Make_Path: Creates a new host view from an existing RCI or R1000 view. This command also creates a new remote directory and a remote Ada program library associated with the new host view.
- Cmvc.Make_Subpath or Rci_Cmvc.Make_Subpath: Creates a new host view from an existing RCI view. This command also creates a remote directory and a remote Ada program library associated with the new host view.

The commands in package Cmvc obtain the names of the remote machine and remote directory through the default switch mechanism described in "Specifying Remote Login Information" on page 20. If you want to override the default switch mechanism, use the commands in package Rci_Cmvc, which allow you to specify these names through the Remote_Machine and Remote_Directory parameters.

Creating Remote Libraries for Existing Views

The extension may provide operations that create or replace Ada directories and libraries on the remote machine when views already exist on the host:

- Rci.Build_Remote_Library: Builds a new remote directory and remote Ada program library for an existing RCI view. Units in the host view are demoted to the installed state; they are downloaded to the remote directory only when you promote them to the coded state on the host.
- Rci.Rebuild_Remote_Library: Destroys and rebuilds the remote directory and remote Ada program library for an existing RCI view. By default, units are downloaded to the new remote directory and compiled.

If supported, the Rci.Rebuild_Remote_Library command is especially useful after a view-creation command from package Cmvc or Rci_Cmvc takes no action on the remote machine because the remote library already exists. When this happens, you can use Rci.Rebuild_Remote_Library to destroy all components of the existing remote library and build a new remote library. The command also downloads the Ada source units to the remote directory and compiles them if you set the Remake-_Demoted_Units parameter to True.

Alternatively, you can destroy a remote library and build a new one in two separate operations by entering the Rci.Destroy_Remote_Library command followed by the Rci.Build_Remote_Library command. Together, these commands execute the same remote commands as the Rci.Rebuild_Remote_Library command. Note, however, that the Rci.Build_Remote_Library command demotes units in the host view to the installed state and therefore does not download them to the new remote directory. You must promote the host-view units to the coded state as a separate operation in order to download them to the new remote directory.

Creating Remote Libraries Manually

The extension provides automatic remote library and project management. Even after the customization is performed, however, there are a number of reasons why you may want to create a remote library manually:

- Specialized mapping structure
- Existing directory structures
- Build failure due to incorrect parameters
- No network access to the remote machine

This subject is covered in "Explicit Creation" in Chapter 6.

The instructions for your remote operating system are:

Destroying Views and Remote Libraries

The extension may provide operations that destroy views and remote libraries:

- Cmvc.Destroy_View: Destroys the remote library (unless the Host_Only library switch for the associated host view has been changed to True) and then destroys the associated RCI host view.
- Rci.Destroy_Remote_Library: Destroys the remote library associated with an RCI host view.

These commands delete the remote library in the remote directory that is specified in the Ftp.Remote_Directory library switch for the specified host view. They delete all the Ada information in the remote directory, including the Ada program library, the downloaded source units, and all compilation information. After deleting all the files in the remote directory, these commands then destroy the directory.

Controlling Host and Remote Imports

Imports are handled through different methods depending on your extension. The following commands control host and remote imports:

- Cmvc.Initial or Rci_Cmvc.Initial
- Cmvc.Make_Path or Rci_Cmvc.Make_Path
- Cmvc.Make_Subpath or Rci_Cmvc.Make_Subpath

- Cmvc.Release or Rci_Cmvc.Release
- Rci.Build_Remote_Library
- Rci.Rebuild_Remote_Library

The Cmvc and Rci import commands may control imports in host views and remote libraries:

- Cmvc.Import: Adds an imported view.
- Rci.Refresh_Remote_Imports: Refreshes the remote imports to match the host imports.
- Cmvc.Remove_Import: Removes an imported view.

Your extension provides the following method for imports:

Compiling and Linking Remote Libraries

When you use the Common.Promote or Compilation.Make command on the host to promote one or more units to the coded state, the extension automatically determines which units have been edited since the last time they were coded, downloads any new or changed units to the remote machine, and compiles them.

Linking is done automatically if the main program you are compiling contains a pragma Main. Otherwise, you can use the Rci.Link command as a separate operation.

Releasing Views and Remote Libraries

The extension provides operations for creating releases of host views and remote libraries. The following commands accomplish this, provided that the Session_Rci-.Auto_Create_Remote_Directory switch has its default value (True):

- Cmvc.Release or Rci_Cmvc.Release: Makes a released view from a host view and creates a corresponding remote directory and Ada program library.

RCI BATCH-COMPILATION SUPPORT

The extension can be customized to support batch-compilation mode, as described in Chapter 4. Under batch-compilation mode, promote and demote operations affect only units on the host; when you are ready to compile units on the remote compilation platform, you use the Rci.Build_Script command to:

- Generate a batch script containing commands that invoke the target compiler for the relevant units
- Optionally download the units to be compiled
- Optionally download the batch script to the remote machine and execute it there

This extension provides the following support for batch mode:

NETWORK-COMMUNICATIONS MECHANISM

The network-communications mechanism controls command and file transfer between the host and the remote compilation platform. The network-communications method for the extension is either Telnet/FTP or DTIA.

- You can use the Compare_Objects parameter for the following commands:
  - Rci.Check_Consistency: Checks whether a unit has been edited remotely since it was last coded on the host (and therefore downloaded).
  - Rci.Accept_Remote_Changes: Uploads changes made remotely into the host unit.

  For extensions that use DTIA, when the Compare_Objects parameter is False, the commands compare the timestamp values for the host and remote files. When the Compare_Objects parameter is True, these commands compare files by uploading and comparing text. For extensions that use FTP, the RCI uploads and compares the file text; the value of Compare_Objects is always considered True.
- You can use the Consistency parameter of the Rci.Show_Units command, which displays the current information about the state and configuration of a specified unit. When the Consistency parameter is True in a DTIA extension, the timestamps of the files are compared for consistency. If it is True in an FTP extension, the RCI uploads and compares the text files.

TROUBLESHOOTING

This section contains troubleshooting techniques for your extension.

D

Quick Reference for Parameter-Value Conventions

Parameters of RCI and CMVC commands accept values that conform both to Ada and to Environment-defined conventions. This Quick Reference summarizes the Environment-defined conventions for parameter values.

WHERE TO LOOK

- Conventions for referencing objects:

- Conventions for specifying other command inputs:

- Explanations and examples of all Environment-defined parameter-value conventions:

  Parameter-Value Conventions tabbed section in the Reference Summary (RS) book of the *Rational Environment Reference Manual*

- Introductory material describing Ada parameter syntax:
  Rational Environment User's Guide

PATHNAMES

- Must be enclosed in quotation marks.
- Consist of name components separated by periods.
  - *Fully qualified pathnames* start with !
    `"!Users.Anderson.Calculation"`
  - *Relative pathnames* start with a character other than ! and are resolved relative to the current library:
    `"!Users.Anderson.Calculation"`
  - *Simple names* contain only one component:
    `"Calculation"`
  - Pathnames of *deleted* objects are enclosed in braces:
    `"{Old_Memo}"`
- Can be abbreviated with wildcards or context characters.

LIBRARY.RESOLVE COMMAND

- Is useful for testing naming expressions before you use them in commands.
- Evaluates special names, context characters, wildcard characters, substitution characters, set notation, indirect file notation, and attributes.
- Displays the fully qualified pathname(s) to which an expression expands, relative to the context in which the command is entered:

```
Library.Resolve (Name_Of => "[@,~lee,~miyata,~chavez]");
```

- Displays the fully qualified pathname(s) to which a pair of source and destination expressions expand (as when used in move/copy operations):

```
Library.Resolve (Name_Of     => "File@",
                 Target_Name => "Old_File@");
```

DESIGNATION

There are three ways to designate an object (specify it for command operation):

- Position the cursor on its name or in its image.
- Select (highlight) its name or image using commands from packages !Commands.Common.Object or !Commands.Editor.Region.
  - Useful key combinations: [Object] - [←]
    [Region] - [[] and [Region] - []]
- Select its name or image *and* position the cursor in the highlighted area.

PARAMETER PLACEHOLDERS

- Appear as default parameter values in many commands.
- Have the form `">>clue<<"` (for example, `">>LIBRARY NAME<<"`).
- Must be replaced with a pathname appropriate to the *clue*.

SPECIAL NAMES

- Are predefined strings that provide a shorthand for referencing various objects.
- Must be enclosed in quotation marks.
- Can be upper- or lowercase.
- Can be abbreviated to shortest unambiguous string within the angle brackets.

Special Names for Specific Objects

Shortest Form	Full Form	Description
"<h>"	"<HOME>"	Resolves to the user's home world.
"<su>"	"<SUBSYSTEM>"	Resolves to the enclosing subsystem.
"<vi>"	"<VIEW>"	Resolves to the enclosing view.

Special Names for Designated Objects

Shortest Form	Full Form	Description
"<c>"	"<CURSOR>"	Resolves to the object on which the cursor is located; any highlighted area is ignored.
"<r>"	"<REGION>"	Resolves to the highlighted object; cursor can be anywhere. Often specifies a highlighted source object in a copy command, where cursor specifies the destination.
"<s>"	"<SELECTION>"	Resolves to the highlighted object; the highlight must contain the cursor or an error results.
"<i>"	"<IMAGE>"	Resolves to the highlighted object containing the cursor (if any); otherwise, resolves to the object whose image contains the cursor.
"<t>"	"<TEXT>"	Resolves to the object whose name is a highlighted string containing the cursor. Equivalent to copying the highlighted string directly into the parameter prompt.

Special Names for Default Objects

Shortest Form	Full Form	Description
"<a>"	"<ACTIVITY>"	Resolves to the highlighted activity containing the cursor (if any); otherwise, resolves to the default activity for the current session.
"<sw>"	"<SWITCH>"	Resolves to the highlighted library switch file; otherwise, resolves to the library switch file associated with the highlighted library; otherwise, resolves to the library switch file associated with the current image. The highlighted area (if any) must contain the cursor.

CONTEXT CHARACTERS

- Are shorthand for object names based on object location or relationship to other known objects.

Context Characters

Character	Description
!	Resolves to the Environment's root world.
!!*name*	Resolves to the Environment's root world on an R1000 called *name* (only for commands in packages Archive and Queue).
[]	Resolves to the current context (either a library or an Ada unit).
$	Resolves to the current library, when used alone or at the beginning of a name. (The current library either is or encloses the current context.
^	Resolves to the closest enclosing object (either a library or an Ada unit), when used alone or at the beginning of a name.
$[*name*]	Resolves to the closest enclosing library whose simple name is *name*.
$$[*name*]	Resolves to the closest enclosing world whose simple name is *name*.
^[*name*]	Resolves to the closest enclosing object whose simple name is *name*.
foo	Evaluates the name *foo* relative to the searchlist for the current session.
library`*foo*	Evaluates the simple name *foo* relative to the links associated with *library*.
unit`*foo*	Evaluates the simple name *foo* relative to the objects that are directly visible in *unit*.

DEBUGGER CONTEXT CHARACTERS

- Are used in debugger commands to reference stack frames, tasks, and Ada units in the program being debugged.
- Can be used in addition to general context characters (see above).

Debugger Context Characters

Character	Description
`.name`	Resolves *name* to an Ada unit that is referenced in the program being debugged. Subsequent name components resolve to objects declared in that Ada unit: `Source (".Initialize.Status")`
`_n`	Refers to the stack frame with the specified frame number (1 is the top frame). Subsequent name components resolve to objects in the subprogram whose activation is contained in the frame: `Put ("_3.Status")`
`%name, %n`	Refers to the task with the specified task name or number. Subsequent name components resolve to objects declared in the task: `Put ("%Debug_Shell._3.Status")`

WILDCARD CHARACTERS

- Match portions of pathnames.
- Allow you to abbreviate single pathnames or specify multiple pathnames as a single string.
- Differ from *restricted naming expressions* and *pattern-matching characters* for search strings (see page 213).

Wildcard Characters

Character	Description
#	Matches a single character: `T####` matches Tools.
@	Matches zero or more characters: `!U@.@.Tools` matches !Users.Anderson.Tools.
?	Matches zero or more nonworld name components: `!Users.Anderson?` matches !Users.Anderson and all objects in it except worlds.
??	Matches zero or more name components, including worlds and their contents: `!Users??` matches !Users, all user home worlds, and their contents.

SUBSTITUTION CHARACTERS

- Are shorthand for specifying destination names in move and copy operations.
- Derive one or more destination names from portions of source names.

Substitution Characters

Character	Description
@	Expands to the string or strings matched by a wildcard (#, @, ?, ??) in the source name; allows you to copy, move, or rename multiple objects in a single operation. The following command renames File1 through File50 to be Old_File1 through Old_File50: `Lib.Rename (From => "File@",  -- wildcard @` `To => "Old_File@"); -- substitution @`
#	Expands to a name component from the source name; saves typing when the source and destination names have components in common. The following command copies !Users.Anderson.Memo to !Users.Anderson.Documentation.Memo: `Lib.Copy (From => "!Users.Anderson.Memo",` `To => "!#.#.Documentation.#");`

SET NOTATION

- Allows you to specify multiple names in a single string.

Set Notation

Notation	Description
`[...]`	Delimits a set of fully qualified or relative names: `"[!users.lee,!users.miyata,!users.chavez]"` or name segments: `"!users[lee,miyata,chavez]tools"`
`[...`*name*`;`*name*`...]`	Separates names in set; requires every name in the set to resolve (an unresolved name is an error): `"[lee;miyata;chavez]"`
`[...`*name*`,`*name*`...]`	Separates names in set; allows unresolved names to be ignored without causing errors: `"[lee,miyata,chavez]"`
`[...~`*name*`...]`	Excludes a name from a set: `"!users[@,~lee,~miyata,~chavez]"`

INDIRECT FILES

- Provide a convenient way to use the same set of pathnames in multiple commands or in commands that are entered multiple times.
- Are specified as a parameter value by prefixing the filename with an underscore:

```
Archive.Save (Objects => "_users_in_my_group");
```

- Contain a list of fully qualified or relative pathnames; can contain *wildcards*, *attributes*, *set notation*, and other indirect files.
  - Names are entered on separate lines or are separated by commas or semicolons (see separators for *set notation*).
  - Names on separate lines are the same as names separated by commas:

```
!users.lee
!users.miyata
!users.chavez
```

  - Command converts contents of indirect file into set notation.

RESTRICTED NAMING EXPRESSIONS

- Are used in parameters that accept only a restricted subset of naming possibilities (the For_Prefix of Archive.Copy, the Source parameter of Links.Display).
  - Such parameters match string names against a list rather than resolving them against the library system.

Restricted Naming Expressions

Character	Description
#	Matches a single character other than a period: `T####` matches Tools.
@	Matches zero or more characters not containing a period: `!U@.@.Tools` matches !Users.Anderson.Tools.
?	Matches zero or more name components of any kind: `!Users.Anderson?` matches !Users.Anderson and everything in it.
`[...]`	Encloses a set of names: `[!Users.Anderson?,!Users.Miyata?]` matches everything in the home worlds for Anderson and Miyata.
~name	Excludes a name from a set: `[@,~Tools]` matches everything except Tools.

PATTERN-MATCHING CHARACTERS

- Are used in regular expression matching for search and comparison commands (packages Editor.Search and File_Utilities).

Wildcard Characters for Pattern Matching

Character	Description
?	Matches any single character.
%	Matches any single character that is legal in an Ada identifier.
$	Matches Ada delimiters: & ' (* + , - . / : ; < = > | When used outside brackets ([]), $ matches beginning and end of line as well.
\	Quotes the next wildcard character, causing it to have a literal (not a wildcard) interpretation. \ must immediately precede the wildcard it quotes.
{	Matches the beginning of a line, when used at the beginning of the pattern; otherwise, { has a literal meaning.
}	Matches the end of a line, when used at the end of the pattern; otherwise, } has a literal meaning.
[]	Defines a set of characters, of which any one can be matched. The set can be a list (for example, [ABCDE]) or a range (for example, [A-Z]).
^	Excludes the next character or set of characters; ^a matches any character other than a, and ^[abc] or [^abc] matches any character other than a, b, or c.
*	Matches zero or more occurrences of the previous character or set of characters.

ATTRIBUTES

- Specify properties of objects.
- Are postfixed to name components in pathname: `Calculation'Body`
- May or may not accept arguments (see tables, pages 216 to 221).
  - Arguments are enclosed in parentheses: `My_File'V(5)`
  - Multiple arguments are separated by commas: `My_File'V(4,6)`
- Are often used with wildcards to specify sets of objects by their properties:
  - Multiple attributes denote properties shared by all matched objects: `@'Body'S(Installed)` matches installed unit bodies.
  - Multiple arguments to a single attribute denote disjoint properties: `@'S(Installed,Coded)` matches either installed or coded Ada units.

- Arguments preceded by ~ denote excluded properties:
  `@'C(~Binary)` matches all objects except binary files.
  `@'C(File'C(~Binary))` matches all files except binary files.

Attributes

Attribute	Description
`'Body`	Resolves to the body of an Ada unit: `"Calculation'Body"`
`'C(arguments)`	Specifies an object's class or subclass: `"@'C(Library)"`
`'If(arguments)`	Specifies whether an object is controlled, checked in, checked out, or frozen: `"@'If(Frozen)"`
`'L(argument)` *or* `'L`	Specifies a library's links for resolution of subsequent name components. Omitting *argument* specifies entire set of links: `"Tools'L.Text_Io"`
`'N(nickname)`	Specifies an Ada subprogram's nickname: `"Example.Overloaded'N(First)"`
`'S(arguments)`	Specifies an Ada unit's compilation state: `"@'S(Source)"`
`'Spec`	Resolves to the visible part (specification) of an Ada unit: `"Calculation'Spec"`
`'Spec_View(activity)` *or* `'Spec_View`	Specifies the spec view listed for a given subsystem in *activity*. Omitting *activity* uses the default activity: `"!Project.Interface.@.Units'Spec_View"`
`'T(target_key)`	Specifies a library's target key: `"@_Working'T(Mc68020_Bare)"`
`'V(arguments)`	Specifies an object's version: `"My_File'V(5)"`
`'View(activity)` *or* `'View`	Specifies the load view listed for a given subsystem in *activity*. Omitting *activity* uses the default activity: `"Command_Interpreter'View"`

Attributes with Predefined Arguments

Arguments for the Conditional Attribute 'If

Short Form	Full Form	Description
Controlled		Matches objects if they are controlled.
In	Checked_In	Matches objects if they are controlled and checked in.
Out	Checked_Out	Matches objects if they are controlled and checked out.
Frozen		Matches objects if they are frozen.

Arguments for the Link Attribute 'L

Argument	Description
Any	Resolves the next name component relative to external and internal links (the default if no argument is specified).
External	Resolves the next name component relative to external links only. External links reference Ada units outside the closest enclosing world.
Internal	Resolves the next name component relative to internal links only. Internal links reference Ada units within the closest enclosing world or any of its subdirectories.

Arguments for the Compilation-State Attribute 'S

Shortest Form	Full Form	Description
A	Archived	Matches units in the archived state.
S	Source	Matches units in the source state.
I	Installed	Matches units in the installed state.
C	Coded	Matches units in the coded state.

Arguments for the Version Attribute 'V

Argument	Description
All	Matches *all* versions of the object.
Any	Matches only the *default* version of the object, which may but need not be the newest version.
Max	Matches the *newest* version of the object, which may but need not be the default version.
Min	Matches the *oldest* retained version of the object.
n	Matches the version with version number *n*. Multiple version numbers can be listed, separated with commas.
-n	Matches the *n*th version preceding the newest version.

Arguments for the Class Attribute 'C: Object Classes

Argument	Description
Ada	Ada program units of any subclass
Archived_Code	Objects appearing in a subsystem view for a code-only unit
File	Files of any subclass
Group	Groups defined for access control
Library	Libraries of any subclass
Null_Device	Devices that accept output and discard it
Pipe	Pipes
Session	User session objects
Tape	Tape drives in the system
Terminal	Terminals in the system
User	Users in the system

Arguments for the Class Attribute 'C: Library Subclasses

Short Form	Full Form	Description
Comb_Ss	Combined_Subsystem	Combined subsystem (can contain only combined views)
Comb_View	Combined_View	Combined view of a subsystem
Directory		Directory
Load_View		Load view of a subsystem
Spec_View		Spec view of subsystem
Subsystem		Spec_Load subsystem (can contain spec, load, or combined views)
System	System_Subsystem	System (object managed using package Cmvc_Hierarchy)
Sys_View	System_View	View in a system
World		World

Arguments for the Class Attribute 'C: Ada Subclasses

Short Form	Full Form	Description
Alt_List	Alternative_List	Insertion point for alternative list
Comp_Unit	Compilation_Unit	Compilation unit that has not been semanticized
Context	Context_List	Insertion point for context clause
Decl_List	Declaration_List	Insertion point for declaration list
Func_Body	Function_Body	Function body
Func_Inst	Function_Instantiation	Generic function instantiation
Func_Ren	Function_Rename	Function rename
Func_Spec	Function_Spec	Function specification
Gen_Func	Generic_Function	Generic function
Gen_Pack	Generic_Package	Generic package
Gen_Param	Generic_Parameter_List	Insertion point for generic parameter
Gen_Proc	Generic_Procedure	Generic procedure
Insertion	Nonterminal	Insertion point
Load_Func	Loaded_Function_Spec	Code-only function
Load_Proc	Loaded_Procedure_Spec	Code-only procedure
Main_Body	Main_Function_Body	Main function body
Main_Body	Main_Procedure_Body	Main procedure body

Arguments for the Class Attribute 'C: Ada Subclasses (continued)

Short Form	Full Form	Description
Main_Func	Main_Function_Spec	Main function specification
Main_Proc	Main_Procedure_Spec	Main procedure specification
Pack_Body	Package_Body	Package body
Pack_Inst	Package_Instantiation	Generic package instantiation
Pack_Ren	Package_Rename	Package rename
Pack_Spec	Package_Spec	Package specification
Pragma	Pragma_List	Insertion point for pragma
Proc_Body	Procedure_Body	Procedure body
Proc_Inst	Procedure_Instantiation	Generic procedure instantiation
Proc_Ren	Procedure_Rename	Procedure rename
Proc_Spec	Procedure_Spec	Procedure specification
Statement	Statement_List	Insertion point for statement
Subp_Body	Subprogram_Body	Subprogram body
Subp_Inst	Subprogram_Instantiation	Generic subprogram instantiation
Subp_Ren	Subprogram_Rename	Subprogram rename
Subp_Spec	Subprogram_Spec	Subprogram specification
Task_Body		Task body

Arguments for the Class Attribute 'C: File Subclasses

Short Form	Full Form	Description
Activity		Activity file (used with subsystem development)
Binary		Binary file
Cmvc_Acc	Cmvc_Access	File containing CMVC access-control information for a view or subsystem
Cmvc_Db	Cmvc_Database	CMVC database (stores source control information in subsystems)
Code_Db	Code_Database	Code saved for a subsystem load view
Compat_Db	Compatibility_Database	Compatibility database for a subsystem
Config	Configuration	Configuration pointer for CMVC
Dictionry	Dictionary	For future development
Documents	Document_Database	Document database (part of Rational Design Facility)

Arguments for the Class Attribute 'C: File Subclasses (continued)

Short Form	Full Form	Description
Elements	Element_Cache	Element cache for storing permanent collections of Ada program elements (part of Rational Design Facility)
Exe_Code	Executable_Code	Executable module generated by the linker of the Cross-Development Facility
File_Map	Pure_Element_File_Map	File map
Log		Log file
Mail		Collections of messages (part of Rational Network Mail)
Mail_Db	Mail_Database	User's mailbox (part of Rational Network Mail)
Markup		Text file containing markup, generated from abstract document (part of Rational Design Facility)
Msg_In	Incoming_Mail_Message	For future development
Msg_Out	Outgoing_Mail_Message	For future development
Obj_Code	Object_Code	Relocatable object module generated by the compilation system of the Cross-Development Facility
Objects	Object_Set	Permanent collection of Directory-.Object
Ps	Postscript	PostScript file
Search	Search_List	Searchlist file
Switch		Switch file
Swtch_Def	Switch_Definition	Switch definition file
Text		Text file
Venture		A collection of work orders for CMVC
Work	Work_Order	Work order for CMVC
Work_List	Work_Order_List	Work-order list for CMVC

OPTIONS PARAMETER

- Accepts one or more *option specifications*:
  `Options => "options specifications"`
  - Option specifications are strings that assign *values* to *options*.
  - A given option can accept values that are Booleans, predefined literals, or user-specified strings (such as pathnames, time expressions, and so on).

Option Specifications

Syntax	Meaning
option = value *or* *option => value* *or* *option := value*	General format for an option specification; assigns *value* to *option* using any of three delimiters: `Options => "After=12/15/9"` *Values* containing commas, semicolons, =, =>, or := must be enclosed in parentheses: `Options => "Label:=(May 26, 1991)"` `Options => "Object_Acl=>(John=>RW)"`
option *or* *option*= true	Specifies Boolean option with value True: `Options => "Replace"` `Options => "Replace => True"`
~option *or* *option*= false	Specifies Boolean option with value False: `Options => "~Replace"` `Options => "Replace := False"`
value *or* *option = value*	Specifies predefined literal value for *option*: `Options => "Fixed_Length"` `Options => "Format = Fixed_Length"`
opt = val, opt = val,... *or* *opt = val; opt = val;...*	General format for multiple option specifications; uses commas or semicolons as separators (with equivalent meaning): `Options =>` `"After=12/15/91, Format=R1000")`
opt1 | opt2 | ... = val	Shorthand format for assigning the same value (*val*) to two or more options: `Options =>` `"Object_Acl|Default_Acl=Retain")`
(~) option (~) option ...	Shorthand format for specifying multiple Boolean options with either True or False values; uses blanks as separators: `Options => "Replace Promote"`

RESPONSE PARAMETER

- Specifies the response characteristics for a command.
  - Response characteristics include a command's error response, log generation, message output and format, activity, remote-passwords file, and remote-sessions file.
- Accepts special values that specify one of the following prepackaged sets of response characteristics:
  - The *system default profile* (provided by Environment for general use)
  - The *session response profile* (defined in current session-switch settings)
  - The *job response profile* (same as session response profile, unless reset for the job using package Profile commands)
- Accepts special values that filter message output.
- Accepts option specifications that tailor individual response characteristics.

Special Values for Specifying Profiles

Special Value	Description
"<PROFILE>"	Causes the command to obtain its response characteristics from the job response profile.
"<SESSION>"	Causes the command to obtain its response characteristics from the session response profile, ignoring the job response profile.
"<DEFAULT>"	Causes the command to obtain its response characteristics from the system default profile, ignoring the job and session response profiles.

Special Values for Filtering Messages

Special Value	Description
"<ERRORS>"	Logs only negative, error, and exception messages (++*, ***, %%%); perseveres at errors, without raising an exception; otherwise, same as job response profile.
"<IGNORE>"	Logs no messages; perseveres at errors, without raising an exception; otherwise, same as job response profile.
"<NIL>"	Logs no messages; quits at the first error, without raising an exception; uses no activity, remote-passwords, or remote-sessions file; ignores job and session response profiles.
"<PROGRESS>"	Logs only positive, negative, error, and exception messages (+++, ++*, ***, %%%); perseveres at errors, without raising an exception; otherwise, same as job response profile.
"<QUIET>"	Same as "<IGNORE>".
"<RAISE_EXCEPTION>"	Raises an exception at the first error and quits immediately; otherwise, same as job response profile. (For package Archive commands, you should use the string "Raise_Error, <PROFILE>" instead to permit graceful termination after exception is raised.)

Special Values for Filtering Messages (continued)

Special Value	Description
"<VERBOSE>"	Logs all messages except debug messages (???); otherwise, same as job response profile.
"<WARN>"	Logs only negative, warning, error, and exception messages (++*, !!!, ***, %%%); perseveres at errors, without raising an exception; otherwise, same as job response profile.

Response Parameter Options

Caution: *Unspecified options are set to nil; to be safe, use options along with special names. Examples:*

```
Response => ("Width=255, <PROFILE>")
Response => ("Use_Error, <PROFILE>")
Response => ("Symbols, <PROFILE>")
```

- `Width = n`

  Sets the width of the message output to the specified number of characters.

- `Reaction = literal`

  Specifies how the command responds to errors:

`Quit`	Stops immediately at the first error; does not raise an exception.
`Propagate`	Stops immediately at the first error; raises an exception.
`Persevere`	Continues processing when an error is encountered; does not raise an exception.
`Raise_Error`	Continues processing when an error is encountered; raises an exception after all processing is complete.

- `Message-symbols`

  Boolean options that control whether the corresponding types of messages appear in the log. There are twelve such options, one for each type of message:

  ```
  :::   ???   ---   +++   >>>   ++*
  !!!   ***   %%%   ###   @@@   $$$
  ```

- `Prefix = literals`

  Specifies up to three fields of information to be prefixed to each message:

  ```
  TIME       DATE        SYMBOLS
  HR_MN_SC   MN_DY_YR
  HR_MN DY_  MON_YR
             YR_MN_DY
  ```

- `Log_File = literal`

  Specifies where log messages are to be directed:

`Use_Output`	Directs messages to Current_Output (by default, same as Standard_Output).
`Use_Error`	Directs messages to Current_Error (by default, same as Standard_Error).
`Use_Standard_Output`	Directs messages to Standard_Output (an Environment output window).
`Use_Standard_Error`	Directs messages to Standard_Error (the Environment message window).

- `Activity = activity_name`

  Specifies the name of the activity to use during execution.
- `Remote_Passwords = filename`

  Specifies the remote-passwords file to use when accessing remote machines.
- `Remote_Sessions = filename`

  Specifies the remote-sessions file to use when accessing remote machines.

Index

A

B

C

D

directory, remote (*continued*)

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by sending e-mail to support@rational.com. If you use e-mail to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____

_____

_____

_____

_____

_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____

_____

_____

_____

Indicate any additions or changes you would like to see in the index.

_____

_____

_____

_____

How much experience have you had with the Rational Environment?

❑ 6 mo. or less ❑ 6 mo.–1 year ❑ 1–3 years ❑ 3 years or more

How much experience have you had with the Ada programming language?

❑ 6 mo. or less ❑ 6 mo.–1 year ❑ 1–3 years ❑ 3 years or more

Name (optional) _____ Date _____

Company _____

Address _____

City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
RATIONAL
3320 Scott Boulevard
Santa Clara, CA 95054-3197

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by sending e-mail to support@rational.com. If you use e-mail to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____

_____

_____

_____

_____

_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____

_____

_____

_____

Indicate any additions or changes you would like to see in the index.

_____

_____

_____

_____

How much experience have you had with the Rational Environment?

❑ 6 mo. or less ❑ 6 mo.–1 year ❑ 1–3 years ❑ 3 years or more

How much experience have you had with the Ada programming language?

❑ 6 mo. or less ❑ 6 mo.–1 year ❑ 1–3 years ❑ 3 years or more

Name (optional) _____ Date _____

Company _____

Address _____

City _____ State _____ ZIP Code _____

Please return this form to: **Publications Department**
RATIONAL
3320 Scott Boulevard
Santa Clara, CA 95054-3197