Rational Environment Reference Manual

Library Management (LM)

Document Control Number: 8001A-25

Rev. 1.0, July 1987 (Delta)

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

RATIONAL

How to Use This Book

The Library Management (LM) book of the *Rational Environment Reference Manual* describes the operations that users can perform on libraries, such as library organization, compilation of large systems, and control of access to libraries and objects in those libraries. It also describes operations for copying, saving, and restoring objects; link operations; library switch operations; operations for comparing, merging, and searching Ada® units and files; and cross-reference lists.

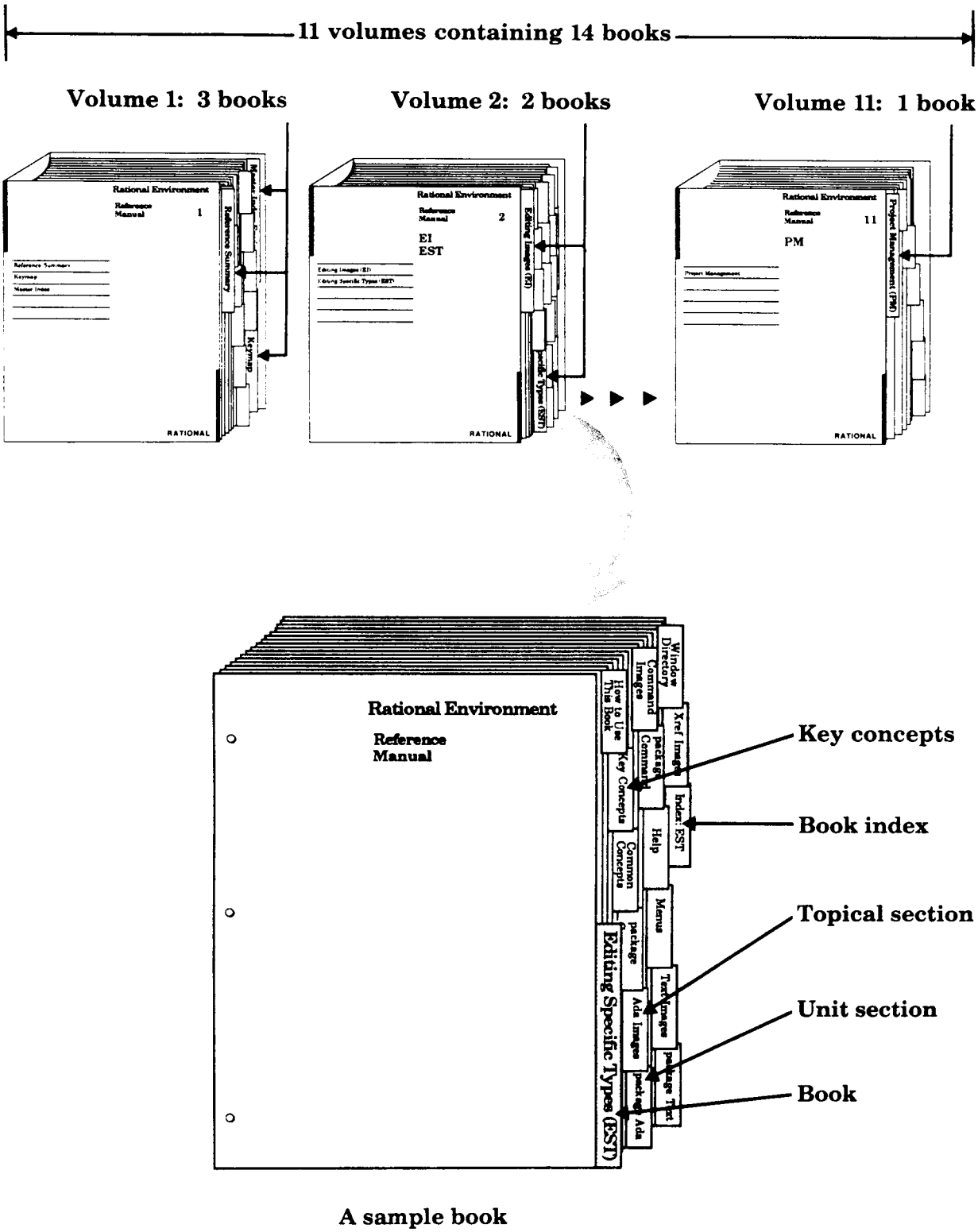
Organization of the Reference Manual

The *Rational Environment Reference Manual* (Reference Manual for brevity) includes the following volumes (see accompanying illustration):

1	Reference Summary Keymap Master Index
2	Editing Images (EI) Editing Specific Types (EST)
3	Debugging (DEB)
4	Session and Job Management (SJM)
5	Library Management (LM)
6	Text Input/Output (TIO)
7	Data and Device Input/Output (DIO)
8	String Tools (ST)
9	Programming Tools (PT)
10	System Management Utilities (SMU)
11	Project Management (PM)

Each *volume* of the Reference Manual contains one or more *books* separated by large colored tabs. Each book contains information on particular features or areas of application in the Environment. The abbreviation for the name of each book (for example, EI for Editing Images) appears on the binder cover and spine, and this abbreviation is used in page numbers and cross-references. The books grouped into one volume are not necessarily logically related.

Organization of the *Rational Environment Reference Manual*



A sample book

The Reference Manual provides reference information organized to efficiently answer specific questions about the Rational Environment. The *Rational Environment User's Guide* complements this manual, providing a user-oriented introduction to the facilities of the Environment. Products other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility) are documented in individual manuals, which are not part of the Reference Manual.

Volume 1

Volume 1, intended to be used as a quick reference to the resources provided by the Environment, contains the following books:

- **Reference Summary**: The Reference Summary contains the full Ada specification for each unit in the standard Environment. The unit specifications are organized by their pathnames. The World ! section provides a list of the units in the library system of the Environment and the manual/book in which they are documented.
- **Keymap**: The Rational Environment Keymap presents the standard Environment key bindings, organized by topic and by command name. The topical section includes both a quick reference for commonly used commands and a more detailed reference for key bindings.
- **Master Index**: The Master Index combines all of the index information for each of the books in the Reference Manual.

Volumes 2–11

Each book in Volumes 2–11 begins with a colored tab on which the name of the book appears. Each book typically contains the following sections:

- **Contents**: The table of contents provides a complete list of all the units in the book and their reference entries.
- **Key Concepts section**: Most of the books contain a section describing key concepts that pertain to all of the Environment facilities documented in that book. This section is located behind its own tab after the table of contents.
- **Unit sections**: Each of the commands, tools, and so on has a declaration within an Ada compilation unit (typically a package) in the Environment library system. For each unit, there is a section that contains reference entries for the declarations (for example, procedures, functions, and types) within that unit. Each section is preceded by a tab.

  The sections for units are alphabetized by the simple names of the units. For example, the section for package !Tools.String_Utilities is alphabetized under String_Utilities.

  For many units, introductory material and/or examples specific to the unit appear after the section tabs.

  Within the section for a given unit, the reference entries describing the unit's declarations are organized alphabetically after the section introduction. Appearing at the top of each page in a reference entry are the simple name of the given declaration and the fully qualified pathname of the enclosing unit.

- **Explanatory/topical sections**: Like the unit sections, explanatory/topical sections are preceded by tabs, and they are alphabetized with the unit sections. The topical sections, such as Help, located in Editing Specific Types (EST), discuss Environment facilities.
- **Index**: Preceded by a tab, the Index appears as the last section of each book. It contains entries for each unit or declaration, along with additional topical references. Each book index covers only the material documented in that particular book. The Master Index (in Volume 1) provides entries for the information documented in all the books within the Reference Manual.

  Italic page numbers indicate the page on which the primary reference entry for a declaration appears; nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Suggestions for Finding Information

The following suggestions may help you in finding various kinds of information in the documentation for Rational's products.

Learning about Environment Facilities

If you are a novice user starting to use the Environment, consult the *Rational Environment User's Guide.*

If you are familiar with the Environment but are interested in learning about the Environment's library-management commands, for example, you might start by scanning the specifications for these units in the Reference Summary to get an idea of the kinds of things these tools can do. You should also look at the Key Concepts for the particular book, which describes important concepts and gives examples.

It may also be useful to glance through the introductions provided for some of the units in the book. These introductions, located immediately after the tabs for the units, often contain helpful examples.

Finding Information on a Specific Item

If you know the name of the item and the book in which it is documented, consult either the table of contents or the index for that book. You can also turn through the pages of the book using the names and pathnames of the reference entries to locate the entry you want. Remember that the reference entries for a unit are organized alphabetically within the unit, and the units are organized alphabetically by simple name within the book.

If you know the simple name of the entry but do not know the book in which it is documented, look in the Master Index (in Volume 1) to find the book abbreviation and page number.

If you know the pathname of the entry but do not know the book in which it is documented, the World ! section of the Reference Summary (in Volume 1) provides a map of the units in the library system of the Environment and the books in which they are documented.

If you cannot find an item in the Master Index, the item either is not documented or is documented in the manuals for a product other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility). If you know the pathname, consult the World ! section of the Reference Summary to determine whether that item is documented and in which manual.

Using the Index

The index of each book contains entries for each unit and its declarations, organized alphabetically by simple name. When using the index to find a specific item, consult the italic page number for the primary reference for that item. Nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Viewing Specifications On-Line

If you know the pathname of a declaration and want to see its specification in a window of the Rational Environment, provide its pathname to the Common.Definition procedure—for example, `Definition ("!Commands.Library");`. If you know the simple name of the unit in which the declaration appears, in most cases you can use searchlist naming as a quick way of viewing the unit—for example, `Definition ("\Library");`.

Using On-Line Help

Most of the information contained in the reference entries for each unit is available through the on-line help facilities of the Environment. Press the Help on Help key or consult the *Rational Environment User's Guide* or the *Rational Environment Reference Manual*, EST, Help, for more information on using this on-line help facility.

Cross-Reference Conventions

The following conventions are used in cross-references to information:

- **Specific page/book**: For references to a specific place in a specific book, the book abbreviation is followed by the page number in the book (for example, LM-322). If the book abbreviation is omitted, the current book is implied (for example, the page numbers in the table of contents for a book do not include the book prefix).
- **Declaration in same unit**: References to the documentation for a declaration in the same unit are indicated by the simple name of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Library.Move procedure would be simply "procedure Move." Note that if there are nested packages in the unit, references to nested declarations use qualified pathnames.
- **Declaration in different unit, same book**: References to the documentation for a declaration in another unit are indicated by the qualified pathname of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Compilation.Delete procedure would be "procedure Compilation.Delete."

- **Declaration in different book**: References to the documentation for a declaration in another book are indicated by the addition of the abbreviation for that book. For example, within the reference entry for the Library.Copy procedure, a reference to the Editor.Region.Copy procedure in the Editing Images book would be "EI, procedure Editor.Region.Copy."

References to specific declarations in the library system of the Rational Environment (not the documentation for them) are typically indicated by fully qualified pathnames—for example, "procedure !Commands.Library.Copy." When the context is clear, however, a shorter name will be used. If the unit in which the declaration appears is undocumented, you may want to see its explanatory comments to understand what it does. To see these comments, either look at the unit's specification in the Reference Summary or view it on-line using the Rational Environment.

Feedback to Rational: Reader's Comments Form

Rational wants to make its documentation as useful and error-free as possible. Please provide us with feedback. The last page of each book contains a Reader's Comments form that you can use to send us comments or to report errors. You can also submit problem reports and make suggestions electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Key Concepts

Managing your libraries suggests several things: organizing libraries, compiling large systems, controlling access to libraries and objects in those libraries, and utilizing Rational Environment resources. This Library Management book of the *Rational Environment Reference Manual* describes several packages that aid in managing your work. These packages are:

- Access_List: Defines a set of operations for interactively displaying, setting, and changing *access lists* (ACLs) and *default access lists* for worlds, Ada units, and files.
- Access_List_Tools: Defines a set of operations for programmatically displaying, setting, and changing access lists and default access lists. Access lists are the mechanism by which access to worlds, Ada units, and files is controlled.
- Archive: Defines a set of operations for copying, saving, and restoring single or multiple objects, as well as for transferring objects back and forth between Rational systems. This package also permits the copying, saving, and restoring of code only for subsystems and main programs.
- Compilation: Defines a set of operations for promoting, demoting, creating, or deleting large programs in which the structure of those programs is not known.
- File_Utilities: Defines a set of operations for comparing, merging, and searching Ada units and files.
- Library: Defines a set of operations for creating, moving, copying, or deleting the objects in the library system. This package also describes the type-specific editing operations available on library images.
- Links: Defines a set of operations for creating, editing, and using *links*. This package also describes the type-specific editing operations available on link images. Links are the Environment mechanism for utilizing Ada units from one library in another library.
- Switches: Defines a set of operations for creating, editing, and manipulating *library switches*. This package also describes the type-specific editing operations available on switch images. Switches provide a means of tailoring specific attributes of the editor, compilation system, pretty-printer, or other Environment facility.
- Xref: Defines a set of operations that generate lists of the Ada units that reference user-selectable Ada constructs in other Ada units.

These packages and their commands are described in detail later in this book of the *Rational Environment Reference Manual.* Some concepts that apply to several of these facilities are described in the remainder of this section.

Library System

The library system in the Environment is a hierarchy of directories and worlds, both of which are referred to as *libraries.* Special attributes, which are attached to worlds to form points for controlling resources, differentiate directories from worlds.

Objects in the hierarchy can have multiple versions. Each version is assigned a number by the Environment when the version is created. Each object has, at most, one current version and possibly several deleted versions. Deleted versions are retained until expunged or until newer versions are created. Objects also can be deleted, but they are retained until they are expunged.

Worlds are closed scope for Ada naming and require that program units in the world that need facilities outside the world explicitly import those needed facilities. These imports are specified in the set of links that are associated with the enclosing world.

Worlds

Worlds are used primarily where the contents of the structure are other structural elements or programs.

The root of the library system is the world "!". The home library of all users is a world. Another common use of worlds is for project-specific libraries.

Worlds have the special attribute that they and their contents are built on the same disk volume.

Worlds also contain the set of links that import facilities for program units in the world or in any directories in the world.

Directories

Directories are used to contain related Ada units, main programs, or other directories or worlds.

Directories behave just as worlds do except that directories do not contain links and they always exist on the same disk volume as their parent world.

Library Editing and Listing

The Rational Environment provides type-specific editing operations on library images. These editing operations are described in more detail in package Library in this book.

The Environment also provides a set of commands for listing the contents of either a directory or a world. These listing commands can provide a large amount of data on individual units.

For example, consider a directory called !Users.Lance.Client_Layer that contains the following units:

```
!Users.Lance.Client_Layer
  Mail_Man
  Mail_Man
  Select_Act
  Select_Act
```

From this view of the directory, several pieces of information about the units in the directory are not available. Editing operations are available to expand the information displayed in the image of this directory. For more information, see the introductory information on the type-specific editing operations available on libraries in package Library. The listing commands can also provide this information.

The listing commands display information about a named or selected unit or units. The information is displayed in Current_Output, which is, by default, an output window. Typically, the command displays information about all of the subunits.

For example, information can be displayed about the units in the above directory with the Library.Ada_List procedure. The output from that command is:

```
------------------------------------------------------------------------------

!USERS.LANCE.CLIENT_LAYER  % LIBRARY.ADA_LIST              STARTED 09:07:59 PM
------------------------------------------------------------------------------


87/06/03 21:08:01 ::: Listing of !USERS.LANCE.CLIENT_LAYER.@'C(ADA)  sorted
87/06/03 21:08:01 ... by declaration.

 STATUS                 DECLARATION
=========  ================================
CODED      Mail_Man : Ada (Pack_Spec);
SOURCE     Mail_Man : Ada (Pack_Body);
INSTALLED  Select_Act     : Ada (Proc_Spec);
SOURCE     Select_Act     : Ada (Proc_Body);
```

This listing provides information about all of the units in the directory, including the class of the unit (in this case, Ada) and the subclass of the unit (for example, package spec, package body, and so on). The listing is sorted alphabetically. Other sorting methods are available.

Another example of information about the same set of objects is displayed with the Library.Verbose_List procedure:

```
--------------------------------------------------------------------------

!USERS.LANCE.CLIENT_LAYER  % LIBRARY.VERBOSE_LIST        STARTED 09:08:30 PM
--------------------------------------------------------------------------

87/06/03 21:08:31 ::: Listing of !USERS.LANCE.CLIENT_LAYER.@'V(ALL)  sorted
87/06/03 21:08:31 ... by object.

      OBJECT      VER  CLASS   SUBCLASS   UPDATER    UPDATE_TIME     SIZE  STATUS
================  ===  =======  ========  =======  ================  ====  =======
MAIL_MAN           2     ADA    PACK_SPEC LANCE  86/06/03 21:06:10   651  SOURCE
                  *3     ADA    PACK_BODY LANCE  86/06/03 21:06:10  2908   CODED
MAIL_MAN'BODY      1     ADA    PACK_BODY LANCE  86/06/03 21:06:22   617  SOURCE
                  *2     ADA    PACK_BODY LANCE  86/06/03 21:06:22  1050  SOURCE
SELECT_ACT         2     ADA    PACK_SPEC LANCE  86/06/03 21:06:41   651  SOURCE
                  *3     ADA    PACK_SPEC LANCE  86/06/03 21:06:41  2899 INSTALLED
SELECT_ACT'BODY    1     ADA    PACK_BODY LANCE  86/06/03 21:06:45   617  SOURCE
                  *2     ADA    PACK_BODY LANCE  86/06/03 21:06:46  1063  SOURCE
```

This listing includes several version numbers for some objects. Objects can have several deleted versions that are retained until they are expunged. The default version is marked with an asterisk (*).

Additional information includes the class of the object. Typically, this is an Ada class, library class, or file class object.

The listing also includes the size, in bytes, of the object. The unit state (STATUS), an indication of the frozen status, and the retention count for the object are also listed. Some or all of these items may appear beyond the right edge of the window.

A number of other fields of information can be displayed with these listing commands, including the username of the person who last updated or read the object and the date and time when that occurred.

Compilation

Compilation management is the control of the compilation of sets of interdependent Ada units. These dependencies between units can cause difficulty in managing the compilation of large systems. The Environment provides several capabilities that make the management of this compilation process much easier.

The Environment maintains a database of dependencies between units. As a unit is created and changed, any dependencies the unit has on other units are recorded. This dependency database allows the Environment to know that, when one unit is changed, the units that depend on the changed unit must be recompiled. This knowledge is also applied in operations such as deleting Ada units.

Tailoring Your Library

You can tailor libraries and Environment behavior in several ways.

Error Reactions

When errors are discovered in a command, the system can respond by:

- Ignoring the error and trying to continue.
- Issuing a warning message and trying to continue.
- Raising an exception and abandoning the operation.

For each job, the Environment maintains in package Profile (SJM) a default action for commands to take if an error occurs. There are commands to specify and display the default error reaction for a job. Regardless of the default error reaction, any error reaction can be specified for any command.

The Environment has *special values* (used as parameters to commands) for which profile it should use when responding to errors in a command. The three most commonly used are "<PROFILE>", "<SESSION_PROFILE>", and "<DEFAULT>", which refer, respectively, to the job response profile, the session response profile, and the system default profile returned by the Profile.Default_Profile function. See SJM, package Profile, for further information on profiles.

Switches

Several commands in package Compilation use *library switches* to govern certain aspects of their execution. These switches govern the use of optimizations or other operations. In general, the Environment supplies default values for these switches, which are appropriate most of the time. There may be times, however, when some of these switches need to be changed.

Switches are maintained in files. There are commands in package Switches for creating, setting, and displaying switch values in these files. These commands and the type-specific editing operations available on switch images are documented with package Switches.

Each user's session also has a set of switches that control Environment behavior on the user's terminal. Called *session switches*, they control window size, scrolling style, and output window formats. There are also commands in package Switches for setting and displaying the session switches. Session switches are documented in the Session and Job Management (SJM) book of the *Rational Environment Reference Manual.*

Several kinds of messages are produced by these commands. Commentary messages are general notes. Warning, error, and exception messages describe a problem that has arisen in the command. Progress messages indicate whether the command is making positive, errorless progress or negative, erroneous progress.

All of these messages are marked with a character sequence that indicates the kind of message it is. This allows the user to scan a log file looking for a particular character

sequence. Several procedures exist in this package to scan for these sequences. Other tools that automatically scan for these sequences can also be built.

There are also commands for inserting user-defined messages into the log file. The content of these messages can be specified to be of any kind.

Logs are generated under the control of the current profile. This profile is manipulated with procedures from package Profile; the logs can then be manipulated with procedures from package Log. Both packages are documented in the Session and Job Management (SJM) book of the *Rational Environment Reference Manual.*

Using Environment Resources

Environment resources are available for use in programs or for executing directly. Resources to be used in a program must be imported via a mechanism called *links.* Resources to be used from a Command window must be accessible via a mechanism called searchlists.

Importing Environment resources to be used in a program requires importing a specific Ada unit. Thus, a link specifies a library unit.

Resolving names in a Command window requires searching in specific directories or worlds. Thus, searchlists contain a list of directories and worlds.

Links

Links are a way for closed-scope worlds to have access to other resources in other worlds. Ada units that are named in a *with* clause of another Ada unit either must be in the enclosing world or must be visible via a link in the enclosing world.

For example, assume that the following procedure has been built in a world called !Users.Rjb:

```
with Io;
procedure Check_Sum is
   ...
end Check_Sum;
```

When this unit is promoted or checked for semantic errors, the Environment first looks for Io in the current world, !Users.Rjb. If it is not there, the links associated with the world are checked for the unit. In that set of links, a link is found for that unit, as shown below:

```
KIND     LINK => SOURCE
=====================
...
EXT:     IO   => !IO.IO
...
```

Note that EXT specified under KIND refers to the kind of link. EXT specifies an external link. The other kinds of links are INT, an internal link, and OBS, an obsolesced link.

It is important to note that only worlds have links. Directories do not have links but use the set of links associated with the enclosing world.

For example, consider the Ada unit described above. In this example, however, the Ada unit exists not in the world !Users.Rjb but in the directory !Users.Rjb.Test. In this case, when the unit is checked for semantic errors or is promoted, the Environment looks in the current directory for a world for it and then in the links of the nearest enclosing world, because there can be no links in a directory. Again, the Environment would find the same link in the links associated with the world !Users.Rjb.

Links are required for units imported from outside the world as well as for units inside the world. If there are several directories inside the world, utilizing a resource in one directory from another directory requires an internal link. Internal links are created by default for each unit created in the world or in the enclosing directories. The automatic creation of these links is controlled by the Create_Internal_Links library switch. For more information on session switches, see package Switches in this book.

Links can also be used to provide a locally shortened or different name for a unit. The local, link name for the unit can be any name. This allows links to be used to rename units imported into the world.

Naming Objects

Many commands in the Environment require a way of *naming* objects in the Environment to move those objects or to perform operations on those objects. The Environment uses two forms of naming: Ada names and string names. Ada names are used in program units or when executing a command. String names are typically used in the parameters to Environment commands.

Ada names are used to call an Environment command in a Command window or to reference an Ada unit in a program. Ada names are the extended Ada names as defined in the *Reference Manual for the Ada Programming Language*. Ada names are used to reference Ada units only. Files, worlds, directories, and other non-Ada units in the Environment cannot be referenced with an Ada name.

String names are used as arguments to commands. These strings are very similar to Ada names, but they can be used to reference any object in the Environment. Also, string names have five important additions: *special names*, *parameter placeholders*, *wildcards*, *special characters*, and *attributes*. The ability also exists to create a set of names using simple set notations and to substitute characters.

Special Names

Special names are used as parameter values for many Environment operations to specify text, objects, and regions. Special names allow you to designate without providing a pathname. They take the form "<*special name*>", where *special name* specifies text, an object, a region, or an activity, as described below. Anywhere that a string name can be used, a special name can be used.

Listed below are the special names used in the Environment and their references:

"<SELECTION>"	References the object associated with the highlighted area, when the cursor is located in the highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the object on which the cursor is located, whether or not there is a highlighted area in the window.
"<IMAGE>"	References the highlighted object, if the cursor is in the highlighted area. If the cursor is not located in the highlighted area, this special name references the image in which the cursor is located.
"<TEXT>"	References the object named in the highlighted text in the image in the window.
"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, this special name references that activity rather than the default activity.

Special names are used as default parameter values to many operations. Users may replace them with another special name or other form of string name, as accepted by that parameter.

Parameter Placeholders

Many Environment commands use parameter placeholders as default values for parameters. They use the form "*>>parameter placeholder<<*". This naming convention is used, as its name suggests, as a placeholder indicating the type of string name that must be entered to replace it. Executing a command containing a parameter placeholder will result in an error. Parameter placeholders include:

```
">>FILE NAME<<"
">>SOURCE NAMES<<"
">>SWITCH<<"
">>SWITCH FILE<<"
">>SWITCHES<<"
">>WORLD NAMES<<"
```

For example, an operation that has the ">>FILE NAME<<" parameter placeholder requires a filename, such as "!Users.John.File_1".

Wildcards

Wildcards allow for both the abbreviation of names and the specifying of several objects with one name. The wildcards are: pound sign (#), *at* sign (@), question mark (?), and double question mark (??).

The Wildcard

The pound sign (#) represents any single identifier character in a name, including the underscore (_). It can be used several times within a single name. For example, F### will match the name Food.

Any wildcard can be used to represent a set of named objects. For example, if there are objects in the directory !Users.Stooges called Larry, Curly, and Moe, a single string, such as !Users.Stooges.####y, can be created to refer to the first two of them.

The Wildcard @

The *at* sign (@) represents zero or more identifier characters in a name, including the underscore (_). It does not match any subunits of Ada units. The wildcard can be used several times within a single name. For example, the name !Users.Fred.Food can be written !U@.@.Food, if that abbreviation is unambiguous.

This wildcard can be used to represent a set of named objects. For example, if there are objects in the directory !Users.Stooges called Larry, Curly, and Moe, a single string, such as !Users.Stooges.@, can be created to refer to all three of them.

This wildcard can be combined with the special characters (discussed later under "Special Characters") to create very short names that represent sets of objects in the current context. As before, if there are three Ada units in the current context called Larry, Curly, and Moe, the string @ can be used to represent all three Ada units, but it will not include their subunits.

The Wildcard ?

The question mark (?) represents zero or more components in a name that are not worlds or objects contained by those worlds. For example, the name !Users.Stooges? represents the Ada units called Larry, Curly, Moe, and any of their subunits.

Also note that periods before and after the wildcard are optional. For example, the name A.?.B is equivalent to the name A?B.

The Wildcard ??

The double question mark (??) represents zero or more components in a name, including worlds or objects contained by those worlds. For example, the name !Users?? represents the home worlds of all users and the contents of those worlds—for example, !Users.Bill?? and everything in his home world, including worlds and the objects within those worlds. As another example, consider that !?? matches all objects in the directory system on a given machine.

Also note that periods before and after the wildcard are optional. For example, the name A.??.B is equivalent to the name A??B.

Substitution Characters

Similar to the way in which wildcard characters can be used to specify a source group of objects, substitution characters can be used to create target names from source names.

The substitution characters and their definitions are described below. Note that if a substitution character is encountered after all segments/wildcards have been exhausted, the characters are replaced by the null string. If the pound sign (#) or the question mark (?) is replaced by the null string, an immediately following period (.) is also elided from the result string.

The Substitution Character

The pound sign (#) is replaced by the next complete (right to left) segment in a name. For example, if there are Ada units in the world !Users.Stooges called Larry, Curly, and Moe, and the user wants to copy them to a world called !Users.Stooges-.New_World, the user can build the target name parameter (from the source name parameter !Users.Stooges) using substitution characters as follows: !#.#.New_World.#.

The Substitution Character @

The *at* sign (@) is replaced by the portion of the current segment that is matched by a wildcard in the source name. If there is more than one wildcard in the segment, a separate @ character is needed in the target to match each one. Matching is performed from right to left. (For the purpose of this matching, @, #, ?, and ?? are considered wildcards.)

For example, there is a world called !Users.Gzc containing files File_1 through File_50. The user wants to rename these objects as My_File_1 through My_File_50. The source name parameter would be "!Users.Gzc.File_@". The target name parameter, using substitution parameters, would be "!#.#.My_File_@".

The Substitution Character ?

The question mark (?) is replaced by successive full segments, working right to left, until the segment for a world is encountered. For example, to copy everything in the world up through the next-level world !Users.Mary to !Users.John, the source string would be !Users.Mary?? and the target string would be !Users.John?.

Special Characters in Names

Special characters can be used in names to specify either relative or absolute contexts or to specify indirect files of names. These special characters apply to names used throughout the Environment.

A special character in a name determines the context in which the remaining portion of the name will be interpreted. A special character of exclamation (!), caret (^), dollar sign ($), double dollar sign ($$), percent (%), underscore (_), period (.), backslash (\), or grave (`) causes explicit interpretations of the remainder of the name, as described below.

Character pairs are also used to enclose a name and to give that name an additional meaning. Character pairs are brackets ([]) and braces ({}), which are also described below.

The Special Character !

The exclamation mark (!) specifies that the context for resolving the remainder of the name should be set to the root of the library system. This creates a ***fully qualified name.*** This character represents the root of the library system in any context.

The Special Character ^

The caret (^) specifies that the context should be set to the immediately enclosing object. The caret permits naming to climb the hierarchy of objects and eventually reach the root of the library system. The caret prefix can be used repeatedly to define the context to be several units above the current context. The parent object of the root of the directory system is itself.

A special use of this character occurs in combination with a bracketed name. A name component of the form ^[some_unit] resolves to the closest containing object whose simple name is Some_Unit. Brackets are normally used for creating sets of objects.

The caret can also be used as a shorthand method for referring to objects in a parent unit. For example, if the current context is !Users.Pete, another user named Joe can be referred to as !Users.Joe or simply ^Joe.

The Special Character $

The dollar sign ($) specifies that the context should be set to the immediately enclosing library. A library is either a directory or a world. If the current context is a library, this character has no effect.

A special use of this character occurs in combination with a bracketed name. A name component of the form $[some_library] resolves to the closest containing library whose simple name is Some_Library.

The Special Character $$

The double dollar sign ($$) specifies that the context should be set to the immediately enclosing world. This is more restrictive than the single dollar sign ($), which is either a world or a directory. If the current context is a world, this character has no effect.

A special use of this character occurs in combination with a bracketed name. A name component of the form $$[some_world] resolves to the closest containing world whose simple name is Some_World.

The Special Character %

The percent (%), used only in the Rational Debugger, can be used only as the first character of a name. It specifies that the next name component is a task name. Task names are either string names assigned to tasks by calls to the !Commands.Debug-.Set_Task_Name or the !Tools.Debug_Tools.Set_Task_Name procedure or task numbers assigned by the Environment. The !Commands.Debug.Task_Display procedure lists all tasks and their names and numbers.

The components of a name that follow the task name are interpreted as objects declared in the named task. If the task name is followed by _*n* (where *n* is a number), the name refers to a stack frame of the named task. Stack frame names are further discussed in "The Special Character _," below.

The Special Character _

The underscore (_) is interpreted as an indirect file prefix when used in some Environment commands. If the first character after the underscore is an alphabetic character, it is assumed to be the first character of the name of a file that contains other names. This provides a way of building lists of objects and referring to that list in a name. It must also be used when specifying an activity file as an indirect file.

The underscore character is also interpreted as a stack frame prefix when used in the Rational Debugger. If the value of an object declared in a subprogram is to be named, the frame on the run-time stack that contains an activation of that subprogram must be named. Renaming is done using the notation *_frame number*. Stack frames are numbered for each task starting at the top with 1. For example, _4 refers to frame number 4 (fourth frame from the top). Frames are alternately numbered from the bottom using negative numbers.

The Special Character .

The period (.) is used both as a name component separator and as a name prefix. As a separator, it is used just as in Ada names to separate components of a name. For example, in the name Commands.Ada, the period separates the two components of the name.

As a prefix character, the period specifies that the first component of the name is a library unit name. This is used only in the Rational Debugger. A second component of the name would be an object declared in the named library unit.

The Special Character \

The backslash (\) specifies that the next name component be evaluated in the current searchlist. For example, a name such as Larry would be evaluated in the current context. However, a name such as \Larry would be evaluated in each of the contexts of the searchlist in turn until all occurrences of the name Larry are found in those contexts. If more than one occurrence is found, a menu showing all occurrences is displayed.

More information about searchlists can be found in "Using Environment Resources," earlier in this section.

The Special Character `

The grave (`) is used to evaluate names using the current context and the set of links associated with the current context. The grave evaluates the name as if it were the name of an Ada unit in a *with* clause of a unit in the library that contains

the current context. For example, the name `Moe resolves to an Ada unit called Moe in the containing library. Moe could be a link to some other library.

This kind of naming does not allow for renamed packages or instances of generic packages or subprograms to be used. This kind of naming does not "look through" renaming declarations.

More information about links can be found in "Using Environment Resources," earlier in this section.

The Special Characters []

Brackets ([]) define a set notation. Sets are created by enclosing a series of name components, separated by commas, in brackets. For example, the name [Larry, Curly, Moe] represents only those three objects in the current context. The semicolon character can also be used to separate name components. Commas and semicolons cannot be mixed. If semicolons are used, each name component in the set must resolve to at least one object. For example, Foo?['C(Lib), 'Spec] matches any component of Foo that is either a library or an Ada spec. Foo[A;B] must match A and B in Foo.

Names can also be excluded from a set with the tilde (~). For example, the name [@, ~Curly] represents all names in the current context except the name Curly.

The special string [] represents the current context, whether that context is a directory, world, Ada unit, or other object.

The Special Characters {}

Braces ({}) denote objects that have been deleted but not expunged as well as objects that have not been deleted. For example, if the object Curly is deleted but not expunged, @ refers only to Larry and Moe, but {@} refers to Larry, Curly, and Moe.

Attributes

Attributes are special strings that specify a restriction on the evaluation of the name. Syntactically like Ada attributes, these strings are a postfix notation that specifies some restriction on the interpretation of the name. Specific versions of an object, specific classes of objects, either the visible part or the body of an Ada unit, or a nickname can be specified with attributes to remove ambiguity or to specify something other than the default interpretation of the name.

Visible Parts and Bodies

Names normally are searched for in both the visible part or the body of the current context. These attributes can restrict the resolution to either the visible part or the body.

Two Ada unit attributes are defined:

- 'body—Any remaining name components specify an object in the body of the named unit.
- 'spec—Any remaining name components specify an object in the visible part of the named unit.

If no attributes are used for a particular name component, the entire unit, visible part and body, is used to resolve any additional name components. This allows names to be created that specify objects not visible through Ada visibility rules.

Version Attributes

Objects in the directory system can have more than one version. It is necessary, therefore, to distinguish which version of the object is desired. By default, the most recent version is used; if some other version is desired, an attribute can be appended to the name to specify a specific version.

Examples of version attributes are Larry'V(2), Curly'V(ALL), or Moe'V(-1). The value in the parentheses can be any of the following:

ALL	Matches *all* versions of the object.
ANY	Matches the *default* version of the object.
MAX	Matches the *newest* version of the object.
MIN	Matches the *oldest* version of the object.
n	Matches the version with that *version number.*
$-n$	Matches the *n*th version preceding the current version. For example, -1 matches the version created just before the current version.

Class Attributes

Objects in the directory system are of different classes and subclasses. A class or subclass attribute can be used to distinguish which class or subclass of objects is being named. By default, a name assumes any class of object.

Examples of class attributes are L@'C(LIBRARY) or Moe?'C(FILE). The value in the parentheses includes the following classes:

ADA	Any Ada program unit.
ARCHIVED_CODE	Objects appearing in a subsystem view for a code-only unit.
FILE	Any file.
GROUP	Any group in the system.
LIBRARY	Any directory, world, or subsystem.
NULL_DEVICE	A device that accepts output and discards it.
PIPE	Any pipe.
SESSION	Any user's session object.
TAPE	Any tape drive in the system.

TERMINAL	Any terminal in the system.
USER	Any user in the system.

There are many subclasses associated with each class. These are described in Tables 1–1, 1–2, and 1–3.

Table 1–1. Library Class

Subclass Name	*Description*
Comb_Ss	Subsystem containing combined view that cannot contain spec or load views (see PM book)
Comb_View	Combined view of a subsystem (see PM book)
Directory	Directory
Load_View	Load view of a subsystem (see PM book)
Mailbox	Library containing Mail and Mail_Db files for the Rational Mail Utility
Spec_Load	Subsystem that cannot contain combined views (see PM book)
Spec_View	Spec view of subsystem (see PM book)
Subsystem	Subsystem (see PM book)
World	World

Link Attributes

The attribute 'L can be used for matching the link name in the set of links associated with a world. For example, My_World'L(My_Link) matches the link named My_Link in the set of links associated with My_World. The link attribute can take an argument of Any, External, Internal, or any prefix of these. Any specifies either external or internal links, External specifies external links (that is, links referencing units outside the enclosing world), and Internal specifies internal links (that is, links referencing objects within the same enclosing world). For example, Your_World'L(Any)A@ matches all links beginning with the letter A in the set of links for Your_World.

Nickname Attributes

Names of subprograms in an Ada unit can be overloaded. A subprogram can be given a unique nickname with the Nickname pragma, which follows the declaration of the subprogram.

An example of nickname attributes is Larry'N(first). The value in parentheses can be any alphanumeric identifier that corresponds to a nickname that has been defined with the Nickname pragma.

Table 1–2. Ada Class

Subclass	*Description*
Alt_List	Alternative list insertion point
Comp_Unit	Compilation unit that has not been semanticized
Context	Context clause insertion point
Decl_List	Declaration list insertion point
Func_Body	Function body
Func_Inst	Generic function instantiation
Func_Ren	Function rename
Func_Spec	Function specification
Gen_Func	Generic function
Gen_Pack	Generic package
Gen_Param	Generic parameter insertion point
Gen_Proc	Generic procedure
Insertion	Insertion point
Load_Func	Code-only function
Load_Proc	Code-only procedure
Main_Body	Main function body
Main_Body	Main procedure body
Main_Func	Main function specification
Main_Proc	Main procedure specification
Pack_Body	Package body
Pack_Inst	Generic package instantiation
Pack_Ren	Package rename
Pack_Spec	Package specification
Pragma	Pragma insertion point
Proc_Body	Procedure body
Proc_Inst	Generic procedure instantiation
Proc_Ren	Procedure rename
Proc_Spec	Procedure spec
Statement	Statement insertion point
Subp_Body	Subprogram body
Subp_Inst	Generic subprogram instantiation
Subp_Ren	Subprogram rename
Subp_Spec	Subprogram specification
Task_Body	Task body

Table 1-3. File Class

Subclass	*Description*
Activity	Activity file (see PM book)
Binary	Binary file
Cmvc_Db	CMVC database (see PM book)
Code_Db	Code saved for a subsystem load view (see package Archive and PM book)
Compat_Db	Compatibility database for a subsystem
Config	Configuration pointer for CMVC (see PM book)
Dictionary	For future development
Documents	Document database (part of Rational Design Facility)
File_Map	File map
Log	Log file
Mail	Collections of messages (part of Rational Mail Utility)
Mail_Db	User's mailbox (part of Rational Mail Utility)
Msg_In	For future development
Msg_Out	For future development
Objects	Object set
Ps	PostScript file (part of Rational Design Facility)
Search	Searchlist file
Switch	Switch file
Swtch_Def	Switch definition file
Text	Text file
Venture	A collection of work orders for CMVC (see PM book)
Work	Work order for CMVC (see PM book)
Work_List	Work order list for CMVC (see PM book)

State Attributes

The attribute 'S can be used for matching Ada units in a particular state. The state attribute can take an argument of Archived, Source, Installed, Coded, or the first letter of any of these. Archived specifies units in the archived state, Source specifies units in the source state, and so on. For example, !Users.John?'S(Coded) specifies all units in the coded state in John's home library.

The Options Parameter

Many of the commands in the Environment have an optional *options specification* in the form of a parameter called Options. The Options parameter accepts different strings, depending on the command specified.

Syntax Rules

The general form of the Options parameter is *option=>value*. *Option* is the name of an option that modifies the way in which an operation behaves. The =>symbol is called a ***value delimiter*** separating the *option* from the *value*. Other permissible value delimiters are the symbols colon equals (:=) and equals (=). For example, in the Archive.Restore procedure, all of the following specifications of the same option are permissible:

```
"AFTER=>12/25/86"
"AFTER:=12/25/86"
"AFTER=12/25/86"
```

If more than one option is to be specified in the Options parameter, the options must be separated by commas (,) or semicolons (;). For example, in the Archive.Restore procedure, the following two options might be used:

```
"AFTER=12/25/86,FORMAT=R1000"
```

String values specified in options that contain comma or semicolon characters must have the string enclosed in parentheses. For example:

```
"LABEL=(MONDAY, JANUARY 26, 1987)"
```

Two or more options that will be assigned the same value can be combined by separating them with the vertical bar (|), with the value delimiter and value following the last option. For example, two access control options from the Archive.Restore procedure that might take the same value could be specified as:

```
"OBJECT_ACL|DEFAULT_ACL=>(JOHN=>COD)"
```

Sequentially enumerated options that will be assigned the same value can be specified by listing only the first and last options, separated by the double dots (..). For example, in package !Tools.Profile, all log messages can be turned off with the option:

```
"Auxiliary_Msg..Dollar_Msg=>False"
```

Boolean Options: A Special Case

For Boolean options, the value delimiter and value are optional. When they are not specified, the value of the Boolean option is true. To make the value false without using the value delimiter and value, it can be preceded with the tilde (~). For example, specification of the REPLACE Boolean option for the Archive.Restore procedure can be done by specifying any of the following:

```
"REPLACE"
"REPLACE=>TRUE"
"REPLACE:=TRUE"
"REPLACE=TRUE"
```

The value can be set to false by using any of the following:

```
"~REPLACE"
"REPLACE=>FALSE"
"REPLACE:=FALSE"
"REPLACE=FALSE"
```

When Boolean options are specified without the value delimiter and value, the options can be separated by spaces only—for example, from the Archive.Restore procedure:

```
"REPLACE PROMOTE"
```

Boolean sequential enumerations can also be specified without the value delimiter and value. Using the earlier example from package Profile, you could specify the option:

```
"~Auxiliary_Msg..Dollar_Msg"
```

Literals in Options: A Special Case

For literals of the form *literal=value*, the *literal* and *value delimiter* are optional. In the Archive.Restore procedure, for example, the option:

```
"FORMAT=R1000"
```

can be specified as:

```
"R1000"
```

Access Control

Access control allows system managers, project leaders, and individual users to specify who has the right to see, change, delete, or create objects. It controls access in operations that can be performed by jobs—both those performed by users directly executing operations and those performed by jobs explicitly initiated by users. Package Access_List provides an interactive set of procedures that display, set, change, and remove access control for worlds, files, and Ada units. Package Access_List_Tools provides a set of programmatic access control operations.

Users, Identities, and Jobs

When a job is initiated, either when a user directly executes a command or when a user explicitly initiates a job, the job has an *identity*. The identity is the name of the user who initiated it. The identity for a job explicitly initiated by the !Commands.Program.Create_Job and !Commands.Program.Run_Job commands can be set in the Options parameter. The identity of any job can be changed with the !Commands.Program.Change_Identity command.

Worlds, files, and Ada units have *access lists* (ACLs) that control who has access to them. Access to these objects by a job is based on the group membership of the identity initiating that job. An identity can be a member of one or more groups. For example, user John is a member of groups John, Public, Network_Public, and Engineering. For further information on groups, see "Groups," below, and SMU, package Operator.

ACL *entries* consist of a group name to be granted access, the => symbol, and the *classes of access* granted to members of that group—for example, "John=>R". Entries for multiple groups must be separated by commas—for example, "John=>R, Public=>RW". A detailed explanation of access classes appears below.

A job is granted access to an object if the identity that initiated it is a member of one of the groups listed among the entries in the object's ACL and the class of access granted is that which the job requires. If the ACL for a particular object does not contain a group to which the identity belongs, the job will not be permitted access to the object. Furthermore, if the group is not granted the class of access that the job requires, the job will not be permitted to perform the operation. For further information on specific access control situations, see "Specific Cases," below.

An Example

An identity John is a member of groups John, Public, and Group_1. John wants to edit an Ada unit called Unit_1. Groups John and Public do not appear on the ACL for that unit, so John is not granted access based on membership in those groups. The third group of which John is a member, called Group_1, has read and write access to that unit. Therefore, the identity John is granted access to that object because of his membership in group Group_1, which has the required write access.

Groups

Groups may have zero or more members. At a minimum, each user is a member of the group defined by his or her username. When a username is created, it is, by default, a member of groups Public and Network_Public. Note that when either of these groups appears on an ACL, in effect all users are given the specified access to that object.

There is also a special group called Privileged whose members can gain access to any object despite its ACL. To gain access to any object, members of this group can execute the !Commands.Operator.Enable_Privileges command. For further information, see SMU, procedure Operator.Enable_Privileges. Username Operator is a member of this group.

Since a user's identity is established at login, a user must log out and log back in again for a new group membership (added with the !Commands.Operator.Add_To_Group command) to be added to the user's identity.

Operator Capability

Members of group Operator and users who have write access to !Machine.Operator_Capability have *operator capability.* Users with operator capability can execute the Environment commands that require this capability. For further information on groups, see SMU, package Operator.

Objects

ACLs apply to three types of objects: worlds, files, and Ada units. More than one group can be granted each class of access. Access classes for worlds differ from those for files and Ada units. These differences are described below.

Objects other than worlds, files, and Ada units do not have ACLs. In particular, directories do not have ACLs. Changes to directories are controlled by the ACL of the enclosing world. Therefore, if a user wants to create a new object in a directory, the user must have create access for the directory's enclosing world. If the user wants to change the ACL for an object in a directory, the user must have owner access to the enclosing world. The access granted to new objects created in directories is the default ACL associated with the directory's enclosing world. In other words, for access control purposes, directories are transparent.

Access Classes for Worlds

The four access classes for worlds are:

- Create: Required to create new objects in the world (and its contained directories).
- Delete: Required to delete the world.
- Owner: Required to change the ACL of an object in the world, change the links in that world, change the compiler switch associations in that world, and freeze/unfreeze objects in that world.
- Read: Required to view the contents of the world or to resolve names within the world.

Worlds have a *default access list* associated with them. The default ACL specifies the access granted to new objects created in that world.

Access Classes for Files and Ada Units

The two access classes for files and Ada units are:

- Read: Required to inspect the current contents of an object.
- Write: Required to change the value of an object or to delete it.

Mixing Access Classes

You can mix access classes in ACLs (for example, "Public=>RWCOD"). The operation will ignore access classes that are not applicable to the object.

Equivalent Access Classes

The access classes delete and write are equivalent. Therefore, when specifying an ACL for a world, specifying write access is the same as specifying delete access. Also, specifying delete access to an Ada unit or file is the same as specifying write access.

How ACLs Are Assigned When Objects Are Created

When new worlds are created, they are assigned the ACL of the containing world. Worlds also have a default ACL that is given to new files and Ada units created within that world. When a new world is created, its default ACL is set to be the same as that of the containing world. The world's ACL and default ACL can be explicitly changed.

A new version of an object inherits the ACL from the previous version of that object.

Specific Cases

This section describes some specific access control situations.

Access Control and Command Execution

For a command to be executed, a user must have read access to all units named in a Command window. Similarly, a job must have read access to all units named by the Options parameter passed to the !Commands.Program.Run_Job, !Commands.Program.Run, and !Commands.Program.Create_Job commands.

Access Control and Compilation

To promote a unit, the identity must have write access to that unit and read access to any unit it *withs*. To promote a unit and its closure, the identity must have write access to each unit whose state must be changed.

To demote a unit, the identity must have write access to that unit. However, the identity need not have write access to any of the units that *with* it.

Access Control and Links

Library objects other than worlds, Ada units, and files do not have ACLs, although access control may affect them. For example, the addition of links is controlled. To add links from a world to a unit, the user must have read access to the unit and owner access to the world. No access is required to units on which that unit might depend.

Access Control and Networking

Users who have Rational Networking—TCP/IP will find that networking operations are governed by access control as are other user-initiated jobs. FTP operations require that the identity specified in the parameter list have the access required to perform that operation. The identity Network_Public can be used to control the access to objects from users working on a remote machine.

When an RPC server is established on a machine, it will have an identity established for it for access control purposes. This identity can be set by the server itself. Remote requests requiring an identity different from that established by the server can be handled by the server setting its identity based on a username and a password included in the request. For the request to complete successfully, this username and password passed to the server must be legal. The operation used to set a program's

identity can then be called by the server. Thus, a server may be granted the same access as any other program run on an R1000. A server can also be made to run in privileged mode. If this is done, the identity change can be made without the correct password.

A server can have only one identity at a time. When a server must process simultaneous requests, the server cannot have multiple identities established for it at the same time. In such cases, the server must start a separate job for each request, with each job having the appropriate identity. The RPC tools provide the required hooks for accomplishing this. See *Rational Networking—TCP/IP* for further information.

Access Control and Searchlists

To resolve a name on a searchlist, the executing job must have read access to the containing world and to the name. If the executing job does not have the required read access to the containing world, the name will not be resolved and will appear to be undefined.

Access Control and Subsystems

To provide access control in large programming projects, subsystem tools are governed by access control as are other user-initiated operations. No access to the activity is required to delete views it contains.

Access Control and Archiving

When an object is archived, the string form of the object's ACL is saved. When the object is restored with the Archive.Restore procedure, an option permits restoration of the original ACL or the substitution of a new ACL.

Access Control and !Commands

Access control affects certain other operations in !Commands. These packages include Archive (documented in this book), Job (SJM), and Daemon, Operator, Queue, Scheduler, System_Backup, and Terminal (all in SMU).

RATIONAL

package Access_List

Access control allows system managers, project leaders, and individual users to specify who has the right to see, change, delete, or create objects. It controls access in operations that can be performed by jobs—both those performed by users directly executing operations and those performed by jobs explicitly initiated by users. Package Access_List provides an interactive set of procedures that display, set, change, and remove access control for worlds, files, and Ada units.

For information on programmatic access control operations, see package Access_List_Tools, also in this book. For information on group operations, see SMU, package Operator.

Users, Identities, and Jobs

When a job is initiated, either when a user directly executes a command or when a user explicitly initiates a job, the job has an *identity*. The identity is the name of the user who initiated the job. The identity for a job explicitly initiated by the !Commands.Program.Create_Job and !Commands.Program.Run_Job commands can be set in the Options parameter. The identity of any job can be changed with the !Commands.Program.Change_Identity command.

Worlds, files, and Ada units have *access lists* (ACLs) that control who has access to them. Access to these objects by a job is based on the group membership of the identity initiating that job. An identity can be a member of one or more groups. For example, user John is a member of groups John, Public, Network_Public, and Engineering. For further information on groups, see "Groups," below, and SMU, package Operator.

ACL *entries* consist of a group name to be granted access, the => symbol, and the *classes of access* granted to members of that group—for example, "John=>R". Entries for multiple groups must be separated by commas—for example, "John=>R, Public=>RW". A detailed explanation of access classes appears below.

A job is granted access to an object if the identity that initiated it is a member of one of the groups listed among the entries in the object's ACL and the class of access granted is that which the job requires. If the ACL for a particular object does not contain a group to which the identity belongs, the job will not be permitted access

to the object. Furthermore, if the group is not granted the class of access that the job requires, the job will not be permitted to perform the operation. For further information on specific access control situations, see "Specific Cases," below.

An Example

An identity John is a member of groups John, Public, and Group_1. John wants to edit an Ada unit called Unit_1. Groups John and Public do not appear on the ACL for that unit, so John is not granted access based on membership in those groups. The third group of which John is a member, called Group_1, has read and write access to that unit. Therefore, the identity John is granted access to that object because of his membership in group Group_1, which has the required write access.

Groups

Groups may have zero or more members. At a minimum, each user is a member of the group defined by his or her username. When a username is created, it is, by default, a member of groups Public and Network_Public. Note that when either of these groups appears on an ACL, in effect all users are given the specified access to that object.

There is also a special group called Privileged whose members can gain access to any object despite its ACL. To gain access to any object, this group can execute the !Commands.Operator.Enable_Privileges command. For further information, see SMU, procedure Operator.Enable_Privileges. Username Operator is a member of this group.

Since a user's identity is established at login, a user must log out and log back in again for a new group membership (added with the !Commands.Operator.Add_To_Group command) to be added to the user's identity.

Operator Capability

Members of group Operator and users who have write access to !Machine.Operator_Capability have *operator capability.* Users with operator capability can execute the Environment commands that require this capability. For further information on groups, see SMU, package Operator.

Objects

ACLs apply to three types of objects: worlds, files, and Ada units. More than one group can be granted each class of access. Access classes for worlds differ from those for files and Ada units. These differences are described below.

Objects other than worlds, files, and Ada units do not have ACLs. In particular, directories do not have ACLs. Changes to directories are controlled by the ACL of the enclosing world. Therefore, if a user wants to create a new object in a directory, the user must have create access for the directory's enclosing world. If the user wants to change the ACL for an object in a directory, the user must have owner access to the enclosing world. The access granted to new objects created in directories is the default ACL associated with the directory's enclosing world. In other words, for access control purposes, directories are transparent.

Access Classes for Worlds

The four access classes for worlds are:

- Create: Required to create new objects in the world (and its contained directories).
- Delete: Required to delete the world.
- Owner: Required to change the ACL of an object in the world, change the links in that world, change the compiler switch associations in that world, and freeze/unfreeze objects in that world.
- Read: Required to view the contents of the world or to resolve names within the world.

Worlds have a *default access list* associated with them. The default ACL specifies the access granted to new objects created in that world.

Access Classes for Files and Ada Units

The two access classes for files and Ada units are:

- Read: Required to inspect the current contents of an object.
- Write: Required to change the value of an object or to delete it.

Mixing Access Classes

You can mix access classes in ACLs (for example, "Public=>RWCOD"). The operation will ignore access classes that are not applicable to the object.

Equivalent Access Classes

The access classes delete and write are equivalent. Therefore, when specifying an ACL for a world, specifying write access is the same as specifying delete access. Also, specifying delete access to an Ada unit or file is the same as specifying write access.

How ACLs Are Assigned When Objects Are Created

When new worlds are created, they are assigned the ACL of the containing world. Worlds also have a default ACL that is given to new files and Ada units created within that world. When a new world is created, its default ACL is set to be the same as that of the containing world. The world's ACL and default ACL can be explicitly changed.

A new version of an object inherits the ACL from the previous version of that object.

Specific Cases

This section describes some specific access control situations.

Access Control and Command Execution

For a command to be executed, a user must have read access to all units named in a Command window. Similarly, a job must have read access to all units named by the Options parameter passed to the !Commands.Program.Run_Job, !Commands.Program.Run, and !Commands.Program.Create_Job commands.

Access Control and Compilation

To promote a unit, the identity must have write access to that unit and read access to any unit it *withs*. To promote a unit and its closure, the identity must have write access to each unit whose state must be changed.

To demote a unit, the identity must have write access to that unit. However, the identity need not have write access to any of the units that *with* it.

Access Control and Links

Library objects other than worlds, Ada units, and files do not have ACLs, although access control may affect them. For example, the addition of links is controlled. To add links from a world to a unit, the user must have read access to the unit and owner access to the world. No access is required to units on which that unit might depend.

Access Control and Networking

Users who have Rational Networking—TCP/IP will find that networking operations are governed by access control as are other user-initiated jobs. FTP operations require that the identity specified in the parameter list have the access required to perform that operation. The identity Network_Public can be used to control the access to objects from users working on a remote machine.

When an RPC server is established on a machine, it will have an identity established for it for access control purposes. This identity can be set by the server itself. Remote requests requiring an identity different from that established by the server can be handled by the server setting its identity based on a username and a password included in the request. For the request to complete successfully, this username and password passed to the server must be legal. The operation used to set a program's identity can then be called by the server. Thus, a server may be granted the same access as any other program run on an R1000. A server can also be made to run in privileged mode. If this is done, the identity change can be made without the correct password.

A server can have only one identity at a time. When a server must process simultaneous requests, the server cannot have multiple identities established for it at the same time. In such cases, the server must start a separate job for each request, with each job having the appropriate identity. The RPC tools provide the required hooks for accomplishing this. See *Rational Networking—TCP/IP* for further information.

Access Control and Searchlists

To resolve a name on a searchlist, the executing job must have read access to the containing world and to the name. If the executing job does not have the required read access to the containing world, the name will not be resolved and will appear to be undefined.

Access Control and Subsystems

To provide access control in large programming projects, subsystem tools are governed by access control as are other user-initiated operations. No access to the activity is required to delete views it contains.

Access Control and Archiving

When an object is saved with procedures in package Archive, the string form of the object's ACL is saved. When the object is restored with the Archive.Restore procedure, an option permits restoration of the original ACL or the substitution of a new ACL.

Access Control and !Commands

Access control affects certain other operations in !Commands. These packages include Archive (documented in this book), Job (SJM), and Daemon, Operator, Queue, Scheduler, System_Backup, and Terminal (all in SMU).

Special Names

Many of the commands in this package have *special names* as default values to parameters requiring names. Anywhere that a string name can be used, a special name can be used. Special names allow you to designate without supplying a pathname. They take the form "*<special name>*", where *special name* specifies a text, object, region, or activity, as described below:

"<SELECTION>"	References the object associated with the highlighted area, when the cursor is located in the highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the object on which the cursor is located, whether or not there is a highlighted area in the window.
"<IMAGE>"	References the highlighted object, if the cursor is in the highlighted area. If the cursor is not located in the highlighted area, this special name references the image on which the cursor is located.
"<TEXT>"	References the object named in the highlighted text in the image in the window.
"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, this special name references that activity rather than the default activity.

You can replace special names with other types of naming expressions, as accepted by that parameter.

Error Response

The commands in this package have a Response parameter that specifies how the command should respond to errors, how to generate logs, and what activities to use. The response profile "<PROFILE>", which many commands use by default, specifies the job response profile. If there is no job response profile, the session response profile ("<SESSION_PROFILE>") is used. If there is no session response profile, the system's default profile ("<DEFAULT>") is used. For further information on profiles, see SJM, package Profile.

subtype Acl

```
subtype Acl is String;
```

Description

Defines the form of access lists (ACLs).

This subtype is a string that represents the group names and the classes of access that each group is allowed. If a group is not explicitly listed in the ACL for an object, that group is granted no access. A maximum of seven entries and 512 characters are allowed in an ACL.

A job is granted access to an object based on the identity of the user who initiates it. For the job to obtain access, the user initiating it must be a member of a group granted the required access listed in the ACL.

The form of an individual entry within the ACL specifies a group name, the => symbol, and the access classes granted to that group—for example, "Phil=>RW". Multiple entries in ACLs must be separated by commas: "Phil=>RW,Bob=>R,-Mary=>W".

Example 1

The following example of an ACL for a world grants members of two groups (GZC and Public) different classes of access to that world. Group GZC has create, owner, and delete access to that world, and group Public has read access only:

```
"GZC=>RCOD,Public=>R"
```

Example 2

The following example of an ACL for a file or an Ada unit grants read and write access to a single group called GZC:

```
"GZC=>RW"
```

procedure Add

```
procedure Add (To_List     : Acl    := "Network_Public => RWCOD";
               For_Object  : Name   := "<SELECTION>";
               Response    : String := "<PROFILE>");
```

Description

Sets the access list (ACL) for the specified object by adding the ACL entry specified in the To_List parameter to the current ACL for the object.

Messages are displayed on Current_Output indicating the success or failure of each operation and the number of objects whose ACLs have been successfully or unsuccessfully set.

Owner access to the world containing the object (or the world itself when changing world ACLs) is required to set ACLs.

Parameters

`To_List : Acl := "Network_Public => RWCOD";`

Specifies the groups for whom access is to be added. The default is to give read, write, create, owner, and delete access to members of group Network_Public.

`For_Object : Name := "<SELECTION>";`

Specifies the object whose ACL will be set. Wildcards, special names, attributes, and context prefixes can be used in specifying the object names. The default is the selected object.

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

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

References

procedure Display

procedure Set

procedure Add_Default

```
procedure Add_Default (To_List    : Acl    := "Network_Public => RW";
                       For_World  : Name   := "<SELECTION>";
                       Response   : String := "<PROFILE>");
```

Description

Sets the default access list (ACL) for the specified world by adding the entry in the To_List parameter to the current default ACL for that world.

New files and Ada units created within the world are given the default ACL. When more than one world is specified by the For_World parameter, messages are displayed on Current_Output indicating the number of worlds whose default ACLs have been successfully or unsuccessfully set. The display also indicates the number of objects that were skipped because they were not worlds.

Parameters

`To_List : Acl := "Network_Public => RW";`

Specifies the group(s) for whom default access is to be granted. The default ACL is given to new objects created within this world. The default gives read and write access to members of group Network_Public.

`For_World : Name := "<SELECTION>";`

Specifies the world whose default ACL is to be set. Wildcards, special names, attributes, and context prefixes can be used in specifying the world names. The default is the selected world. Nonworld objects do not have a default ACL; therefore, when wildcards, special names, attributes, and context prefixes are used to specify worlds, nonworld objects specified by the For_World parameter will be skipped.

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

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

References

procedure Default_Display

constant Create

```
Create : constant Character := 'C';
```

Description

Defines an access class that applies only to worlds.

This access class permits the user to create new objects in the specified world, with the exception of worlds contained within that world.

See "References," below, for other access classes.

References

constant Delete

constant Owner

constant Read

constant Write

procedure Default_Display

```
procedure Default_Display (For_World : Name := "<CURSOR>");
```

Description

Displays the default access list (ACL) for the world specified.

New objects created within a world are given the default ACL. If the selected object is not a world and "<CURSOR>" or "<SELECTION>" is displayed as a parameter, no display will result.

Parameters

```
For_World :  Name := "<CURSOR>";
```

Specifies the worlds whose default ACLs will be displayed. The default is the world on which the cursor is currently located. Wildcards, special names, and context prefixes can be used in specifying the world names. When wildcards are used, nonworld objects will not be displayed, because they do not have default ACLs.

Example

The command:

```
access_list.default_display (for_world=>"!users.gzc");
```

produces a display indicating that users in group GZC have read and write access to new objects created within that world as shown in the following display:

```
------------------------------------------------------------------------
!USERS.GZC % ACCESS_LIST.DEFAULT_DISPLAY                STARTED 6:38:22 PM
------------------------------------------------------------------------

!USERS.GZC : GZC=>RW
```

References

procedure Set_Default

---

constant Delete

```
Delete : constant Character := 'D';
```

Description

Defines an access class that applies only to worlds.

This access class, which permits deletion of the specified world, is synonymous with write access.

See "References," below, for other access classes.

References

constant Create

constant Owner

constant Read

constant Write

procedure Display

```
procedure Display (For_Object : Name := "<CURSOR>");
```

Description

Displays the access list (ACL) for the specified object on Current_Output.

Objects that have ACLs are worlds, files, and Ada units.

Parameters

```
For_Object :  Name := "<CURSOR>";
```

Specifies the object whose ACL will be displayed. The default is the object on which the cursor is currently located. Wildcards, special names, attributes, and context prefixes can be used in specifying the object names.

Names in the same context are factored on the display (see "Example 2," below).

Example 1

Assuming that the object on which the cursor is located is a file called File_1, the command:

```
access_list.display (for_object=>"!users.czg.file_1");
```

produces the following display in Current_Output. Assuming that Current_Output is a window, this display indicates that users in group CZG have been granted read and write access to File_1.

```
------------------------------------------------------------------------
!USERS.CZG % ACCESS_LIST.DISPLAY                    STARTED 6:34:17 PM
------------------------------------------------------------------------

!USERS.CZG.FILE_1  : CZG=>RW
```

Example 2

The command:

```
access_list.display (for_object=>"!users.gzc??");
```

creates a list of the ACLs in Current_Output. Because a number of worlds are involved, the resulting display is factored by worlds, as shown below:

```
-------------------------------------------------------------------------------

!USERS.GZC % ACCESS_LIST.DISPLAY                          STARTED 6:56:13 PM
-------------------------------------------------------------------------------

Context: !USERS
GZC                              : GZC=>OCD,PUBLIC=>C
GZC.DIRECTORY_1                  : This object has no ACL
GZC.MY_FILE_1'V(1)               : GZC=>RW,NETWORK_PUBLIC=>R
GZC.MY_FILE_2'V(2)               : GZC=>RW,NETWORK_PUBLIC=>R
GZC.MY_S_1'V(1)                  : This object has no ACL
GZC.S_1'V(1)                     : This object has no ACL
GZC.S_1_SWITCHES'V(1)            : GZC=>RW,NETWORK_PUBLIC=>R
GZC.S_2'V(1)                     : This object has no ACL
GZC.TEXT_1'V(3)                  : GZC=>RW,NETWORK_PUBLIC=>R
GZC.UNIT_1'V(4)                  : GZC=>RW,NETWORK_PUBLIC=>R
GZC.UNIT_1'BODY'V(7)             : GZC=>RW,NETWORK_PUBLIC=>R
GZC.UNIT_12'V(3)                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.UNIT_12'BODY'V(3)            : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1                      : GZC=>OCD,PUBLIC=>C
GZC.WORLD_1.DIRECTORY_1          : This object has no ACL
GZC.WORLD_1.SUB_WORLD            : GZC=>OCD,PUBLIC=>C
GZC.WORLD_1.SUB_WORLD.UNIT_1'V(1)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.SUB_WORLD.UNIT_1'BODY'V(1)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.S_1'V(2)             : This object has no ACL
GZC.WORLD_1.TEXT_1'V(2)          : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.UNIT_1'V(4)          : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.UNIT_1'BODY'V(4)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.UNIT_12'V(2)         : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.UNIT_12'BODY'V(2)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.WORLD_1              : GZC=>OCD,PUBLIC=>C
GZC.WORLD_1.WORLD_1.DIRECTORY_1
                                 : This object has no ACL
GZC.WORLD_1.WORLD_1.S_1'V(1)
                                 : This object has no ACL
GZC.WORLD_1.WORLD_1.TEXT_1'V(1)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.WORLD_1.UNIT_1'V(1)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.WORLD_1.UNIT_1'BODY'V(1)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
GZC.WORLD_1.WORLD_1_UNIT'V(3)
                                 : GZC=>RW,NETWORK_PUBLIC=>R
```

References

procedure Set

---

subtype Name

```
subtype Name is String;
```

Description

Defines the subtype for names of objects used by procedures in this package.

This subtype allows all wildcards, special names, context prefixes, attributes, and substitution characters. See the Key Concepts in this book for more general information about naming.

constant Owner

```
Owner  : constant Character := 'O';
```

Description

Defines an access class that applies only to worlds.

This access class permits:

- Changing the access list (ACL) of objects in the specified world.
- Changing the links in the specified world.
- Changing the compiler switch file associations in the specified world.
- Freezing and unfreezing objects in the specified world.

When a world is created, the world's ACL is set to be the same as the ACL of the containing world. A user with owner access is permitted to change the ACL of objects within the world. More than one identity can have owner access.

See "References," below, for other access classes.

References

constant Create

constant Delete

constant Read

constant Write

constant Read

```
Read : constant Character := 'R';
```

Description

Defines an access class that applies to worlds, Ada units, and files.

This access class is required to inspect the contents of an object, including worlds, and to perform operations such as executing the !Commands.Common.Definition command to inspect the contents of an object, opening for In_File mode, and executing certain Rational Debugger commands.

For worlds, this access class allows the user to display the world (or directories therein) and to resolve names in the world (or directories therein).

See "References," below, for other access classes.

Errors

In I/O operations, read access is required for In_File mode. In other words, if a user wants to open a file to read it, the user must have read access to the file. The !Io.Io_Exceptions.Use_Error exception is raised for access failures from Io packages.

References

constant Create

constant Delete

constant Owner

constant Write

procedure Set

```
procedure Set (To_List     : Acl    := "Network_Public => RWCOD";
               For_Object  : Name   := "<SELECTION>";
               Response    : String := "<PROFILE>");
```

Description

Sets the access list (ACL) for the specified object.

Messages are displayed on Current_Output indicating the success or failure of each operation and the number of objects whose ACLs have been successfully or unsuccessfully set.

Owner access to the world containing the object (or the world itself when changing world ACLs) is required to set ACLs.

If a call to the Set procedure attempts to set the ACL of a world, and the user executing the procedure does not have owner access to the world but does have owner access to the enclosing world, the ACL is still set. This allows the user to change the ACL of a world when no one has owner access to it. This exception to the ownership rule applies only to setting the ACL of a world.

Parameters

`To_List : Acl := "Network_Public => RWCOD";`

Specifies the group(s) for whom access is to be set. The default is to give read, write, create, owner, and delete access to members of group Network_Public.

`For_Object : Name := "<SELECTION>";`

Specifies the object whose ACL will be set. Wildcards, special names, attributes, and context prefixes can be used in specifying the object names. The default is the selected object.

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

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

Example

The command:

```
access_list.set
         (to_list=>"gzc=>rw",for_object=>"text",
          response=>"<profile>");
```

changes the ACL for the file named Text, so that the only group to have read or write access to that file is GZC. The confirming display appears as follows:

```
------------------------------------------------------------------------------

!USERS.GZC % ACCESS_LIST.SET                              STARTED 6:40:55 PM
------------------------------------------------------------------------------


86/12/09 18:40:57 --- Access_List.Set (To_List => "GZC=>RW", For_Object =>
86/12/09 18:40:57 ... "<SELECTION>");.
86/12/09 18:40:57 +++ !USERS.GZC.TEXT'V(1):  acl set to GZC=>RW.
```

References

procedure Add

procedure Display

---

procedure Set_Default

```
procedure Set_Default (To_List   : Acl    := "Network_Public => RW";
                       For_World : Name   := "<SELECTION>";
                       Response  : String := "<PROFILE>");
```

Description

Sets the default access list (ACL) for the specified world.

New files and Ada units created within the world are given the default ACL.

When more than one world is specified by the For_World parameter, messages are displayed on Current_Output indicating the number of worlds whose default ACLs have been successfully or unsuccessfully set. The display also indicates the number of objects that were skipped because they were not worlds.

Parameters

`To_List : Acl := "Network_Public => RW";`

Specifies the group(s) for whom default access is to be granted. The default access list is given to new objects created within this world. The default gives read, write, create, owner, and delete access to members of group Network_Public.

`For_World : Name := "<SELECTION>";`

Specifies the world whose default ACL is to be set. Wildcards, special names, attributes, and context prefixes can be used in specifying the world names. The default is the selected world. Nonworld objects do not have a default ACL; therefore, when wildcards, special names, attributes, and context prefixes are used to specify worlds, nonworld objects specified by the name will be skipped.

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

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

Example

Assuming that the selected object is a world, the command:

```
access_list.set_default
          (to_list=>"gzc=>rw",for_world=>"<selection>",
           response=>"<PROFILE>");
```

changes the ACL for that world, so that the only group to have read or write access to new objects created in that world is GZC. The confirming display appears as:

```
------------------------------------------------------------------------------
!USERS.GZC % ACCESS_LIST.SET_DEFAULT                      STARTED 6:42:20 PM
------------------------------------------------------------------------------

86/12/09 18:42:22 --- Access_List.Set_Default (To_List => "GZC=>RW",
86/12/09 18:42:22 ... For_World => "<SELECTION>");.
86/12/09 18:42:22 +++ !USERS.GZC: default acl set to GZC=>RW.
```

References

procedure Add_Default

procedure Default_Display

constant Write

```
Write  : constant Character := 'W';
```

Description

Defines an access class applying to files and Ada units.

This class of access is required to execute operations that change the value of an object, such as editing, promoting, and demoting. Write access is also required to delete objects. Write access is synonymous with delete access.

See "References," below, for other access classes.

Errors

In I/O operations, write access is required for Inout_File and Out_File modes. In other words, if a user wants to open a file for Read/Write or Write mode, the user must have write access to the file. The !Io.Io_Exceptions.Use_Error exception is raised for access failures from Io packages.

References

constant Create

constant Delete

constant Owner

constant Read

end Access_List;

package Access_List_Tools

Access control allows system managers, project leaders, and individual users to specify who has the right to see, change, delete, or create objects. It controls access in operations that can be performed by jobs—both those performed by users directly executing operations and those performed by jobs explicitly initiated by users. Package Access_List_Tools provides a programmatic set of procedures that display, set, change, and remove access control for worlds, files, and Ada units.

For information on interactive access control operations, see package Access_List, also in this book. For information on group operations, see SMU, package Operator.

Users, Identities, and Jobs

When a job is initiated, either when a user directly executes a command or when a user explicitly initiates a job, the job has an *identity*. The identity is the name of the user who initiated it. The identity for a job explicitly initiated by the !Commands.Program.Create_Job and !Commands.Program.Run_Job commands can be set in the Options parameter. The identity of any job can be changed with the !Commands.Program.Change_Identity command.

Worlds, files, and Ada units have *access lists* (ACLs) that control who has access to them. Access to these objects by a job is based on the group membership of the identity initiating that job. An identity can be a member of one or more groups. For example, user John is a member of groups John, Public, Network_Public, and Engineering. For further information on groups, see "Groups," below, and SMU, package Operator.

ACL *entries* consist of a group name to be granted access, the => symbol, and the *classes of access* granted to members of that group—for example, "John=>R". Entries for multiple groups must be separated by commas—for example, "John=>R,-Public=>RW". A detailed explanation of access classes appears below.

A job is granted access to an object if the identity that initiated it is a member of one of the groups listed among the entries in the object's ACL and the class of access granted is that the job requires. If the ACL for a particular object does not contain a group to which the identity belongs, the job will not be permitted access to the object. For further information on specific access control situations, see "Specific Cases," below.

An Example

An identity John is a member of groups John, Public, and Group_1. John wants to edit an Ada unit called Unit_1. Groups John and Public do not appear on the ACL for that unit, so John is not granted access based on membership in those groups. The third group of which John is a member, called Group_1, has read and write access to that unit. Therefore, the identity John is granted access to that object because of his membership in group Group_1, which has the required write access.

Groups

Groups may have zero or more members. At a minimum, each user is a member of the group defined by his or her username. When a username is created, it is, by default, a member of groups Public and Network_Public. Note that when either of these groups appears on an ACL, in effect all users are given the specified access to that object.

There is also a special group called Privileged whose members can gain access to any object despite its ACL. To gain access to any object, this group can execute the !Commands.Operator.Enable_Privileges command. For further information, see SMU, procedure Operator.Enable_Privileges. Username Operator is a member of this group.

Since a user's identity is established at login, a user must log out and log back in again for new group membership (added with the !Commands.Operator.Add_To_Group command) to be added to the user's identity.

Operator Capability

Members of group Operator and users who have write access to !Machine.Operator_Capability have *operator capability*. Users with operator capability can execute the Environment commands that require this capability. For further information on groups, see SMU, package Operator.

Objects

ACLs apply to three types of objects: worlds, files, and Ada units. More than one group can be granted each class of access. Access classes for worlds differ from those for files and Ada units. These differences are described below.

Objects other than worlds, files, and Ada units do not have ACLs. In particular, directories do not have ACLs. Changes to directories are controlled by the ACL of the enclosing world. Therefore, if a user wants to create a new object in a directory, the user must have create access for the directory's enclosing world. If the user wants to change the ACL for an object in a directory, the user must have owner access to the enclosing world. The access granted to new objects created in directories is the default ACL associated with the directory's enclosing world. In other words, for access control purposes, directories are transparent.

Access Classes for Worlds

The four access classes for worlds are:

- Create: Required to create new objects in the world (and its contained directories).
- Delete: Required to delete the world.
- Owner: Required to change the ACL of an object in the world, change the links in that world, change the compiler switch associations in that world, and freeze/unfreeze objects in that world.
- Read: Required to view the contents of the world or to resolve names within the world.

Worlds have a *default access list* associated with them. The default ACL specifies the access granted to new objects created in that world.

Access Classes for Files and Ada Units

The two access classes for files and Ada units are:

- Read: Required to inspect the current contents of an object.
- Write: Required to change the value of an object or to delete it.

Mixing Access Classes

You can mix access classes in ACLs (for example, "Public=>RWCOD"). The system will ignore access classes that are not applicable to the object.

Equivalent Access Clauses

The access classes delete and write are equivalent. Therefore, when specifying an ACL for a world, specifying write access is the same as specifying delete access. Also, specifying delete access to an Ada unit or file is the same as specifying write access.

How ACLs Are Assigned When Objects Are Created

When new worlds are created, they are assigned the ACL of the containing world. Worlds also have a default ACL that is given to new files and Ada units created within that world. When a new world is created, its default ACL is set to be the same as that of the containing world. The world's ACL and default ACL can be explicitly changed.

A new version of an object inherits the ACL from the previous version of that object.

Specific Cases

This section describes some specific access control situations.

Access Control and Command Execution

For a command to be executed, a user must have read access to all units named in a Command window. Similarly, a job must have read access to all units named by the Options parameter passed to the !Commands.Program.Run_Job, !Commands.Program.Run, and !Commands.Program.Create_Job commands.

Access Control and Compilation

To promote a unit, the identity must have write access to that unit and read access to any unit it *withs*. To promote a unit and its closure, the identity must have write access to each unit whose state must be changed.

To demote a unit, the identity must have write access to that unit. However, the identity need not have write access to any of the units that *with* it.

Access Control and Links

Library objects other than worlds, Ada units, and files do not have ACLs, although access control may affect them. For example, the addition of links is controlled. To add links from a world to a unit, the user must have read access to the unit and owner access to the world. No access is required to units on which that unit might depend.

Access Control and Networking

Users who have Rational Networking—TCP/IP will find that networking operations are governed by access control as are other user-initiated jobs. FTP operations require that the identity specified in the parameter list have the access required to perform that operation. The identity Network_Public can be used to control the access to objects from users working on a remote machine.

When an RPC server is established on a machine, it will have an identity established for it for access control purposes. This identity can be set by the server itself. Remote requests requiring an identity different from that established by the server can be handled by the server setting its identity based on a username and a password included in the request. For the request to complete successfully, this username and password passed to the server must be legal. The operation used to set a program's identity can then be called by the server. Thus, a server may be granted the same access as any other program run on an R1000. A server can also be made to run in privileged mode. If this is done, the identity change can be made without the correct password.

A server can have only one identity at a time. When a server must process simultaneous requests, the server cannot have multiple identities established for it at the same time. In such cases, the server must start a separate job for each request, with each job having the appropriate identity. The RPC tools provide the required hooks for accomplishing this. See *Rational Networking—TCP/IP* for further information.

Access Control and Searchlists

To resolve a name on a searchlist, the executing job must have read access to the containing world and to the name. If the executing job does not have the required read access to the containing world, the name will not be resolved and will appear to be undefined.

Access Control and Subsystems

To provide access control in large programming projects, subsystem tools are governed by access control as are other user-initiated operations. No access to the activity is required to delete views it contains.

Access Control and Archiving

When an object is saved with procedures in package Archive, the string form of the object's ACL is saved. When the object is restored with the Archive.Restore procedure, an option permits restoration of the original ACL or the substitution of a new ACL.

Access Control and !Commands

Access control affects certain other operations in !Commands. These packages include Archive (documented in this book), Job (SJM), and Daemon, Operator, Queue, Scheduler, System_Backup, and Terminal (all in SMU).

subtype Access_Class

```
subtype Access_Class is String;
```

Description

Defines the form of access classes.

Access classes consist of the following individual characters (either upper- or lowercase) or combinations thereof: R, W, D, C, and O. The characters stand, respectively, for the following classes of access: read, write, delete, create, and owner.

References

constant Create

constant Delete

constant Owner

constant Read

constant Write

exception Access_Tools_Error

```
Access_Tools_Error : exception;
```

Description

Defines the exception raised when an error condition occurs as the result of executing a function in this package.

When an error occurs as the result of executing a procedure in this package, an error condition and message describing the error are returned in the Status parameter.

subtype Acl

```
subtype Acl is String;
```

Description

Defines the form of access lists (ACLs).

This subtype is a string that represents the group names and the classes of access that each group is allowed. If a group is not explicitly listed in the ACL for an object, that group is granted no access. A maximum of seven entries and 512 characters are allowed in an ACL.

A job is granted access to an object based on the identity of the user who initiates it. For the job to obtain access, the user initiating it must be a member of a group listed in the ACL.

The form of an individual entry within the ACL specifies a group name, the => symbol, and the access classes granted to that group—for example "Phil=>RW". Multiple entries in ACLs must be separated by commas: "Phil=>RW,Bob=>R,-Mary=>W".

Example 1

The following example of an ACL for a world grants members of two groups (GZC and Public) different classes of access to that world. Group GZC has read, create, owner, and delete access to that world, and group Public has create access only:

```
"GZC=>RCOD,Public=>C"
```

Example 2

The following example of an ACL for a file or an Ada unit grants read and write access to a group called GZC:

```
"GZC=>RW"
```

function Amend

```
function Amend (Initial_Acl  : Acl;
                New_Group    : Name;
                Desired      : Access_Class) return Acl;
```

Description

Returns an amended access control list (ACL).

If a username is already granted access because of membership in a group in the initial ACL, the new group will not be appended to the initial ACL.

If the addition of a new entry to the ACL causes the ACL to exceed the maximum of seven entries, the rightmost entry will be removed.

Parameters

`Initial_Acl : Acl;`
Specifies the initial ACL, before it is amended.

`New_Group : Name;`
Specifies the new group to be added to the ACL. Context prefixes and wildcards can be used to specify a single group.

`Desired : Access_Class;`
Specifies the desired access classes for the new group. Access classes consist of either the upper- or lowercase form of one or more of the following: R (read), W (write), C (create), O (owner), or D (delete).

`return Acl;`
Returns an amended ACL.

Errors

The Access_Tools_Error exception is raised if an error occurs.

Common errors include an illegal ACL for the Initial_Acl parameter and an illegal name for the New_Group parameter.

Example

The following example illustrates how the Amend function can be used in conjunction with other procedures in this package to update the ACL for an object:

```
with Log, Simple_Status, Access_List_Tools;
procedure Amend_Example
                (Filename : String;
                 Added_Group : String;
                 Desired_Group_Access : String) is
    Status : Simple_Status.Condition;
begin
...

-- Add a group to the access list, supplied by the user.
    Access_List_Tools.Set (Filename,
                           (Access_List_Tools.Amend
                                 (Access_List_Tools.Get (Filename),
                                  Added_Group,
                                  Desired_Group_Access)),
                            Status);
    Log.Put_Condition (Status);
-- Display the results of adding the group.
...
end Amend_Example;
```

References

subtype Acl

function Get

procedure Get

procedure Set

function Check

```
function Check (User_Name : String := "";
                Object_Id : Directory.Version;
                Desired   : Access_Class) return Boolean;

function Check (User_Name   : String := "";
                Object_Name : String;
                Desired     : Access_Class) return Boolean;

function Check (User_Id   : Directory.Version;
                Object_Id : Directory.Version;
                Desired   : Access_Class) return Boolean;

function Check (Job       : Machine.Job_Id;
                Object_Id : Directory.Version;
                Desired   : Access_Class) return Boolean;
```

Description

Returns true if the specified user or job has the desired access to the specified object.

The value true is also returned if an object that does not have an access list is referenced.

This function returns false if an error is detected during the test or if the job does not have the desired access.

Parameters

`User_Name : String := "";`

Specifies the username whose access class is to be checked. All of the groups to which the username is a member are used for the test. Context prefixes and wildcards can be used to specify a single username. Use of the null string specifies the identity of the calling job.

`User_Id : Directory.Version;`

Specifies the directory version of the user identifier.

`Job : Machine.Job_Id;`

Specifies the directory version of the job identifier.

```
Object_Id :  Directory.Version;
```

Specifies the directory version of the object.

```
Object_Name :  String;
```

Specifies the name of an object. Context prefixes, wildcards, and attributes can be used to specify a single object.

```
Desired :  Access_Class;
```

Specifies the desired access classes. Access classes consist of either the upper- or the lowercase version of one or more of the following: R (read), W (write), C (create), O (owner), or D (delete).

```
return Boolean;
```

Returns true if the specified user or job has the desired access to the specified object or if an object that does not have an access list is referenced. This function returns false if an error is detected during the test or if the job does not have the desired access.

Errors

The Access_Tools_Error exception is raised if an error occurs.

Common errors include illegal values for the Desired parameter and references to objects that do not exist.

Example

The following example shows a simple procedure that uses the Check function to determine whether a user has the desired access to a particular object:

```
with Io, Access_List_Tools;
procedure Check_Example
             (User : String;
              Which_Object : String;
              What_Access : String) is
begin

if Access_List_Tools.Check
         (User, Which_Object, What_Access) then
    Io.Put ("The user has the requested access.");
else
    Io.Put ("The user does not have the requested access.");
end if;
end Check_Example;
```

References

subtype Access_Class

---

procedure Check_Validity

```
procedure Check_Validity (For_List :        Acl;
                          Status   : in out Simple_Status.Condition);
```

Description

Checks the validity of the specified access list (ACL).

Parameters

`For_List : Acl;`
Specifies the ACL to be checked for validity.

`Status : in out Simple_Status.Condition;`
Returns a condition indicating that the procedure has executed correctly. If there is an error, this procedure returns a message indicating the type of error.

Errors

If an error occurs, this procedure returns the Status parameter containing a condition and a message indicating the type of error.

Example

The following example procedure takes as a parameter the name of an object whose ACL is to be checked for validity. If there is an error in the ACL, a display indicates the error. If there is no error, a display indicates that the ACL is valid.

```
with Log, Io, Access_List_Tools, Simple_Status;
procedure Check_Validity (Which_Objects_Access_List  : String) is
Archived
    Status : Simple_Status.Condition;
begin
    Access_List_Tools.Check_Validity
         (Access_List_Tools.Get (Which_Objects_Access_List),  Status);
    if Simple_Status.Error (Status) then
        -- display the error generated by the Simple_Status package.
        Log.Put_Condition (Status);
    else
        Io.Put_Line ("The access list is valid.");
    end if;
end Check_Validity;
```

References

subtype Acl

PT, type Simple_Status.Condition

---

constant Create

```
Create : constant Character := 'C';
```

Description

Defines an access class that applies only to worlds, which permits the user to create new objects anywhere in the specified world.

See "References," below, for other access classes.

References

constant Delete

constant Owner

constant Read

constant Write

constant Delete

```
Delete : constant Character := 'D';
```

Description

Defines an access class that applies only to worlds, which permits deletion of the specified world.

This access class is synonymous with write access.

See "References," below, for other access classes.

References

constant Create

constant Owner

constant Read

constant Write

function Get

```
function Get (For_Object : Name) return Acl;

function Get (For_Object : Directory.Version) return Acl;
```

Description

Returns the access list (ACL) for the specified object.

Objects that have ACLs are worlds, files, and Ada units.

Parameters

`For_Object : Name;`
Specifies the name of the object whose ACL is to be retrieved. Context prefixes, wildcards, and attributes can be used to specify a single object.

`For_Object : Directory.Version;`
Specifies the directory version of the object whose ACL is to be retrieved.

`return Acl;`
Returns the ACL for the specified object.

Errors

The Access_Tools_Error exception is raised if an error occurs. If the user wants to determine which error has occurred, the procedural version of the function should be called. It returns a Status parameter indicating the error.

Example

The following example shows a simple procedure that copies an ACL from one object to another. It uses the Get function to retrieve the ACL of the object whose ACL is to be copied (specified in the Object_To_Get_Acl_From parameter) and then uses the Set procedure to set the ACL (specified in the Object_To_Set_Acl_For parameter).

```
with Log, Simple_Status, Access_List_Tools;
procedure Copy_Acl (Object_To_Get_Acl_From : String;
                    Object_To_Set_Acl_For : String) is
    Status : Simple_Status.Condition;
begin

    Access_List_Tools.Set (Object_To_Set_Acl_For,
                           (Access_List_Tools.Get
                                (Object_To_Get_Acl_From)), Status);
    if Simple_Status.Error (Status) then
        Log.Put_Condition (Status);
    end if;
end Copy_Acl;
```

References

exception Access_Tools_Error

subtype Acl

procedure Get

procedure Set

---

procedure Get

```
procedure Get (For_Object :        Name;
               List       :    out Bounded_String.Variable_String;
               Status     : in out Simple_Status.Condition);

procedure Get (For_Object :        Directory.Version;
               List       :    out Bounded_String.Variable_String;
               Status     : in out Simple_Status.Condition);
```

Description

Gets the access list (ACL) for the specified object and returns it in the List parameter.

Parameters

`For_Object : Name;`

Specifies the name of the object whose access list is to be retrieved. Context prefixes, wildcards, and attributes can be used to specify a single object.

`For_Object : Directory.Version;`

Specifies the directory version of the object.

`List : out Bounded_String.Variable_String;`

Returns the ACL for the specified object. The constraint for this parameter is the Max_Acl_Length constant. If the ACL returned does not fit in the List parameter, the resulting ACL is truncated and the Status parameter is set to indicate this error.

`Status : in out Simple_Status.Condition;`

Returns a condition indicating that the procedure has executed correctly. If there is an error, this procedure returns a condition and a message indicating the type of error.

Errors

If an error is encountered, a condition and a message indicating the type of error are returned in the Status parameter.

Example

The following example shows a simple procedure that uses the procedural form of the Get procedure to retrieve the ACL of a file. It then sets the ACL of a second file to the ACL that has been retrieved.

```
with Log, Io, Simple_Status, Bounded_String, Access_List_Tools;
procedure Copy_Acl
                (Object_To_Get_Acl_From  : String; Object_To_Set_Acl_For
                  : String) is
    Status : Simple_Status.Condition;
    List   :
        Bounded_String.Variable_String (Access_List_Tools.Max_Acl_Length);
begin
    Access_List_Tools.Get (Object_To_Get_Acl_From, List, Status);
    if Simple_Status.Error (Status) then
        Log.Put_Condition (Status);
    else

        Access_List_Tools.Set
            (Object_To_Set_Acl_For, Bounded_String.Image (List), Status);
        if Simple_Status.Error (Status) then
            Log.Put_Condition (Status);
        else
            Io.Put_Line ("The access list has been set for "&
                         Object_To_Set_Acl_For);
        end if;
    end if;
end Copy_Acl;
```

References

function Get

constant Max_Acl_Length

procedure Set

PT, package Simple_Status

function Get_Default

```
function Get_Default (For_World : Name) return Acl;
```

Description

Returns the default access list (ACL) for new objects created in the specified world.

Parameters

For_World : Name;
Specifies the world whose default ACL will be returned. Context prefixes, wildcards, and attributes can be used to specify a single object.

Errors

The Access_Tools_Error exception is raised if an error occurs. If the user wants to determine which error occurred, the procedural form of the Get_Default function returns an error condition and a message in the Status parameter.

Example

The following sample program shows the use of the Get_Default function to retrieve the default ACL for new objects created in a world. The default is then used to set the default ACL for new objects created in another world.

```
with Log, Io, Simple_Status, Access_List_Tools;
procedure Copy_Default_Example
                (World_To_Get_Acl_From  : String; World_To_Set_Acl_For
                   : String) is
    Status : Simple_Status.Condition;
begin

    Access_List_Tools.Set_Default
       (World_To_Set_Acl_For,
        (Access_List_Tools.Get_Default  (World_To_Get_Acl_From)),
        Status);
    if Simple_Status.Error (Status) then
        Log.Put_Condition (Status);
    else
        Io.Put_Line ("The ACL has been set for world "& World_To_Set_Acl_For);
    end if;
end Copy_Default_Example;
```

References

exception Access_Tools_Error

subtype Acl

procedure Get_Default

constant Max_Acl_Length

---

procedure Get_Default

```
procedure Get_Default (For_World :        Name;
                       List      :    out Bounded_String.Variable_String;
                       Status    : in out Simple_Status.Condition);
```

Description

Gets the default access list (ACL) for new objects created in the specified world.

Parameters

`For_World : Name;`

Specifies the world whose default ACL will be displayed. Context prefixes, wildcards, and attributes can be used to specify a single object.

`List : out Bounded_String.Variable_String;`

Returns the default ACL for new objects that are created within the specified world. The constraint for this parameter is the Max_Acl_Length constant. If the ACL returned does not fit in the List parameter, the resulting ACL is truncated and the Status parameter is set to indicate this error.

`Status : in out Simple_Status.Condition;`

Returns a condition indicating that the procedure has executed correctly. If there is an error, the procedure returns a condition and a message indicating the type of error.

Errors

If an error is encountered, a condition and a message indicating the type of error are returned in the Status parameter.

Example

The following example shows how the procedural form of the Get_Default procedure can be used to retrieve the default ACL of new objects created within a world. The default ACL is then used to set the default ACL for new objects created in another world.

```
with Log, Io, Simple_Status, Access_List_Tools, Bounded_String;
procedure Copy_Default_Acl
              (World_To_Get_Acl_From : String;
               World_To_Set_Acl_For  : String) is
   Status : Simple_Status.Condition;
   List   : Bounded_String.Variable_String (Access_List_Tools.Max_Acl_Length);
begin
    Access_List_Tools.Get_Default (World_To_Get_Acl_From, List, Status);
    if Simple_Status.Error (Status) then
       Log.Put_Condition (Status);
    else
       Access_List_Tools.Set_Default (World_To_Set_Acl_For,
       Bounded_String.Image (List), Status);
    end if;
    if Simple_Status.Error (Status) then
       Log.Put_Condition (Status);
    else
       Io.Put_Line ("The default ACL for new objects is set for world"
                    & World_To_Set_Acl_For);
    end if;
end Copy_Default_Acl;
```

References

subtype Acl

function Get_Default

PT, package Simple_Status

ST, package Bounded_String

---

function Has_Operator_Capability

```
function Has_Operator_Capability return Boolean;
```

Description

Returns true if the calling job has operator capability; otherwise, the function returns false.

To have operator capability, the identity initiating the job must meet one of the following conditions:

- Be a member of group operator.
- Have write access to !Machine.Operator_Capability.
- Be a member of group Privileged and have privileges enabled.

Example

The following example shows a simple procedure that determines whether the identity initiating the job has operator capability:

```
with Io, Access_List_Tools;
procedure Has_Operator_Capability_Example  is
begin
    if Access_List_Tools.Has_Operator_Capability  then
        Io.Put ("This job has operator capability.");
    else
        Io.Put ("This job does not have operator capability.");
    end if;
end Has_Operator_Capability_Example;
```

References

SMU, package Operator

constant Max_Acl_Length

```
Max_Acl_Length : constant := 512;
```

Description

Specifies the maximum length in characters for access lists (ACLs).

This constant is used as a constraint for the List parameter in the Get and Get_Default procedures.

subtype Name

```
subtype Name is String;
```

Description

Defines the name of objects used in procedures in this package.

This subtype allows special names, wildcards, context prefixes, and attributes for specification of a single object. See the Key Concepts in this book for more general information about naming.

function Normalize

```
function Normalize (Initial_Acl : Acl) return Acl;
```

Description

Returns a revised access list (ACL).

This function scans the ACL entries for groups that do not currently exist and removes them.

Parameters

`Initial_Acl : Acl;`
Specifies the ACL to be normalized.

`return Acl;`
Returns a revised ACL.

Errors

The Access_Tools_Error exception is raised if an error occurs.

Example

The following simple procedure illustrates how the Normalize function might be used in conjunction with other procedures and functions in this package to normalize an object's ACL. This procedure could be used by the owner of a world after the deletion of several groups that had access to Object_To_Normalize_Acl_For. The owner would use this procedure to remove the deleted groups from the ACL of an object.

```
with Log, Io, Access_List_Tools, Simple_Status;
procedure Normalize_Example (Object_To_Normalize_Acl_For  : String) is
    Status : Simple_Status.Condition;
begin
    Access_List_Tools.Set (Object_To_Normalize_Acl_For,
                           Access_List_Tools.Normalize
                             (Access_List_Tools.Get
                             (Object_To_Normalize_Acl_For)),
                              Status);
    if Simple_Status.Error (Status) then
        Log.Put_Condition (Status);
    else
        Io.Put_Line ("The access list for ");
        Io.Put_Line (Object_To_Normalize_Acl_For);
        Io.Put_Line (" has been normalized.");
    end if;
end Normalize_Example;
```

References

subtype Acl

---

constant Owner

```
Owner : constant Character := 'O';
```

Description

Defines an access class that applies only to worlds.

This access class permits:

- Changing the access list (ACL) of objects in the specified world.
- Changing the links in the specified world.
- Changing the compiler switch file associations in the specified world.
- Freezing and unfreezing objects in the specified world.

When a world is created, the world's access ACL is set to be the same as the ACL of the containing world. A user with owner access is permitted to change the ACL of objects within the world. More than one identity can have owner access.

See "References," below, for other access classes.

References

constant Create

constant Delete

constant Read

constant Write

constant Read

```
Read : constant Character := 'R';
```

Description

Defines an access class that applies to worlds, Ada units, and files.

This access class is required to inspect the current state of an object, including worlds, and to perform operations such as executing the !Commands.Common-.Definition command to inspect the contents of an object, opening for In_File mode, and executing certain Rational Debugger commands.

For worlds, this access class allows the user to display the world (or directories therein) and to resolve names in the world (or directories therein).

See "References," below, for other access classes.

Errors

In I/O operations, read access is required for In_File mode. In other words, if a user wants to open a file to read it, the user must have read access to the file. The !Io.Io_Exceptions.Use_Error exception is raised for access failures from Io packages.

References

constant Create

constant Delete

constant Owner

constant Write

procedure Set

```
procedure Set (For_Object :        Name;
               To_List    :        Acl;
               Status     : in out Simple_Status.Condition);

procedure Set (For_Object :        Directory.Version;
               To_List    :        Acl;
               Status     : in out Simple_Status.Condition);
```

Description

Sets the access list (ACL) for the specified object.

Owner access to the world is required to set ACLs for objects in that world.

If a call to the Set procedure attempts to set the ACL of a world, and the user executing the procedure does not have owner access to the world but does have owner access to the enclosing world, the ACL is still set. This allows the user to change the ACL of a world when no one has owner access to it. This exception to the ownership rule applies only to setting the ACL of a world.

Parameters

`For_Object : Name;`

Specifies the object whose ACL will be displayed. Context prefixes, wildcards, and attributes can be used to specify a single object.

`For_Object : Directory.Version;`

Specifies the directory version of the object.

`To_List : Acl;`

Sets the new ACL for the object.

`Status : in out Simple_Status.Condition;`

Returns a condition indicating that the procedure has executed correctly. If there is an error, this procedure returns a condition and a message indicating the type of error.

Errors

If an error is encountered, a condition and a message indicating the type of error are returned in the Status parameter.

Example

The following example shows a simple procedure that uses the Set procedure to copy the ACL from one object to another. It uses the Get function to retrieve the ACL of the object whose ACL is to be copied.

```
with Log, Simple_Status, Access_List_Tools;
procedure Set_Example (Object_To_Get_Acl_From  : String;
                       Object_To_Set_Acl_For  : String) is
    Status : Simple_Status.Condition;
begin
    Access_List_Tools.Set (Object_To_Set_Acl_For,
                           (Access_List_Tools.Get (Object_To_Get_Acl_From),
                            Status);
    if Simple_Status.Error (Status) then
        Log.Put_Condition (Status);
    end if;
end Set_Example;
```

References

procedure Amend

procedure Get

PT, package Simple_Status

---

procedure Set_Default

```
procedure Set_Default (For_World :        Name;
                       To_List   :        Acl;
                       Status    : in out Simple_Status.Condition);
```

Description

Sets the default access list (ACL) for new objects created in the specified world.

This procedure does not modify the ACLs of already existing objects whose ACLs were set by the previous default ACL.

New files and Ada units created within the world are given the default ACL.

Parameters

`For_World : Name;`
Specifies the world whose default ACL is to be set. Context prefixes, wildcards, and attributes can be used to specify a single object.

`To_List : Acl;`
Sets the new default ACL for new objects created in this world.

`Status : in out Simple_Status.Condition;`
Returns a condition indicating that the procedure has executed correctly. If there is an error, this procedure returns a condition and a message indicating the type of error.

Errors

If an error is encountered, a condition and a message indicating the type of error are returned in the Status parameter.

Example

The following example shows how the Set_Default procedure can be used in conjunction with the Get_Default procedure to retrieve the default ACL of new objects created within a world and to use it to set the default ACL for new objects created in another world.

```
with Log, Io, Simple_Status, Access_List_Tools, Bounded_String;
procedure Get_Default_Procedure_Example
                (World_To_Get_Acl_From  : String; World_To_Set_Acl_For
                   : String) is
    Status : Simple_Status.Condition;
    List   :
        Bounded_String.Variable_String  (Access_List_Tools.Max_Acl_Length);
begin
    Access_List_Tools.Get_Default  (World_To_Get_Acl_From, List, Status);
    if Simple_Status.Error  (Status) then
        Log.Put_Condition  (Status);
    else
        Access_List_Tools.Set_Default  (World_To_Get_Acl_From,
                                        World_To_Set_Acl_For,
                                        Status);
        if Simple_Status.Error  (Status)  then
            Log.Put_Condition  (Status);
        else
            Io.Put  ("The default ACL for new objects is set for world "
                     & World_To_Set_Acl_For);
        end if;
    end if;
end Get_Default_Procedure_Example;
```

References

procedure Get_Default

---

constant Write

```
Write : constant Character := 'W';
```

Description

Defines an access class that applies to files and Ada units.

This class of access is required for operations that change the value of an object, such as editing, promoting, and demoting. Write access, also required to delete objects, is synonymous with delete access.

See "References," below, for other access classes.

Errors

In I/O operations, write access is required for Inout_File and Out_File modes. In other words, if a user wants to open a file for Read/Write or Write mode, the user must have write access to the file. The !Io.Io_Exceptions.Use_Error exception is raised for access failures from Io packages.

References

constant Create

constant Delete

constant Owner

constant Read

end Access_List_Tools;

RATIONAL

package Archive

Overview

The procedures in package Archive are used for archiving and restoring single or multiple objects and for copying objects on the same or different Rational machines.

The Archive facilities differ from system backups in that they can be used selectively to save and restore specific objects, as well as to move objects from one machine to another.

Package Archive contains the following four procedures:

- Copy: Copies one or more objects from one location to another, either on the same R1000 or between two R1000s that are connected through Rational Networking—TCP/IP. This procedure also permits changing access lists (ACLs) and many other options.
- List: Produces a listing of the names of the objects on a tape or library generated by the Save procedure.
- Restore: Reads some or all objects from a tape (or library) generated by the Save procedure and rebuilds the original hierarchical structure from which they were saved. For Ada units, the Restore procedure optionally promotes the units to the states they were in when they were saved. This procedure also permits changing the ACLs of restored objects.
- Save: Writes one or more objects onto a tape (or library), preserving hierarchical structure. For Ada units, the unit state of each unit is also recorded, as well as other information including its ACL, the last updating user, and the last updating time. This procedure also allows the changing of access lists.

The Archive procedures can handle objects such as worlds, directories, Ada units, text files, data files, switch files, activities, subsystems, and subsystem views. The Archive procedures also handle user binary files, transferring the bits unaltered and uninterpreted.

Objects can be moved between R1000s by saving objects to tape with the Save procedure on one R1000 and then using the Restore procedure on the other R1000 to read the tape. This method is recommended for moving a large number of objects.

For moving fewer objects, or for copying objects into a different location on the same R1000, the Copy procedure is a convenient, one-step alternative that does not require a tape or an intermediate library, but it does require Rational Networking—TCP/IP.

The Procedures and Their Parameters

The procedures in this package have parameters that allow you to control the way objects are copied, saved, restored, and listed. All of the procedures in this package take an Options parameter, which allows you to specify additional information to control, for example, which objects are saved and restored, their state when they are restored, and their ACLs. The following sections describe how the parameters and options specified in the Options parameters operate together.

Save

The Save procedure allows you to save a set of objects so that they can later be restored using the Restore procedure. The Objects parameter for the Save procedure allows you to specify which objects are saved. Additionally, the Save procedure takes Options parameters. The Options parameter, among other things, allows you to specify certain attributes that can be used in selecting objects to be saved with the After, CDB, and Nonrecursive options.

The Save procedure allows you to specify where you want to save objects—in a library or on tape—using the Device parameter. The options in the Options parameter that control the way the objects are written to tape are Format, Label, and Version.

Finally, Save takes the Prefix option in the Options parameter, which allows you to specify a naming pattern to be saved with objects. This allows the restorer to restore objects without knowing their names.

For Ada units, files, and worlds, the following information is saved along with them: ACLs, last update time, last updating user, retention count, unit state (Ada units only), and whether the object is frozen. Switch file associations and links are also saved along with worlds.

When you save objects, the Save procedure writes all objects into a single file called Data. A file called Index is also created, which describes the name of each object in the Data file, the size of the object, and its unit state if it is an Ada unit. The Data and Index files can be written either into a library or onto a tape. When the Restore procedure reads these files from the library or tape, they use the Index file to rebuild objects from the Data file and place them in the appropriate states.

The Index and Data files can also be written onto separate tapes. This should always be done when saving large amounts of information that will not fit on one tape.

Restore

The Restore procedure, similar to the Save procedure, has an Objects parameter that is used to specify the objects to be restored. The objects to be restored can be everything saved or a subset of the objects saved. The Objects parameter in the Restore procedure can use pattern matching. Also like the Save procedure, it has a Device parameter that specifies where the objects to be restored are located—on tape or in a library.

The Restore procedure has two parameters, Use_Prefix and For_Prefix, that allow you to control the names under which the objects specified in the Objects parameter will be restored.

Like the Save procedure, the Restore procedure takes an Options parameter. Two options specified in the Options parameter, Overwrite and Replace, can be used to control restoration over existing objects.

The Options parameter for the Restore procedure takes another option, called Promote, that controls whether Ada units will be restored to the same compilation state that they were in when they were archived.

Similar to the Save procedure, the Options parameter in the Restore procedure takes the Format and Label options, which specify the tape format and label. The Restore procedure also takes the CDB and Nonrecursive options, which specify the objects that are to be restored.

The Restore procedure takes four more options in the Options parameter that permit the user to specify the ACLs for the restored objects: Become_Owner, Default_Acl, Object_Acl, and World_Acl.

Finally, the Options parameter for the Restore procedure takes two more options, Primary and Revert_CDB, which are described in "Compatibility Databases, Primaries, and Secondaries," below.

The following list describes what is restored with each type of object:

- Text file: The exact contents of the file are restored.
- Binary file: The exact contents of the file are restored.
- Activity file: When an activity file is restored, the load views and spec views must exist or an error will result.
- Ada unit: When an individual unit is restored, the source is copied and restored in, at least, the source state. If the Promote option is used, the Environment will try to promote it.
- World: When a world is restored, the Environment attempts to restore the links, switch file associations, and ACLs. If the units to which links refer do not exist, errors will occur and the links will not be created. If the switch file associations do not exist, they will not be created.
- User world: When a user world is restored, a corresponding user is created if the restorer has operator capability. The password is the same as the saved user's password. In either case, the contents of the world are restored.

Copy

The Copy procedure should be viewed as a Save followed by a Restore that uses Rational Networking as the medium. It takes the parameters and options required by the Save and Restore procedures, but not those associated with tapes.

Compatibility Databases, Primaries, and Secondaries

Subsystems are built based on the assumption that development of a subsystem is done in one place, called the *primary* subsystem. A copy of the primary that is made with package Archive is called a *secondary* subsystem. A primary can have any number of secondaries. Development cannot take place in a secondary.

To allow you to execute code against both primaries and secondaries, subsystems maintain a *compatibility database* that contains the information required to maintain execution compatibility. With package Archive you can move a primary subsystem to a secondary subsystem and, as part of the move, move the required compatibility database.

With the Primary option, for the Restore and Copy procedures, you can move a primary subsystem and allow the moved copy to be a primary. This is done when the development "home" of a subsystem is to be moved. The original primary should be frozen or deleted. The Primary option is also used when restoring a subsystem from an archive used as a backup.

The Revert_Cdb option allows you to copy an older version of the compatibility database over a newer version. You might need to do this to revert back to an earlier version of a subsystem than the one currently on the machine.

The Options Parameter

As described above, the procedures in this package take an Options parameter. For further information on specifying options, see the Key Concepts in this book. Table 4–1 indicates the options available for each of the procedures in this package. The meaning of each option is described in the reference entry for each command in the Options parameter description.

Examples

This section contains examples that illustrate the use of the operations in this package. It is recommended that you read the reference entries for each of the commands in this package before reading these examples.

Assume that the following world is used in all of the examples shown below:

```
!Users.Phil.Tools  : World;
  Cg_Switches                : C Pack_Spec 87/03/12 12:40:51 Phil     7263  ;
  Cg_Switches                : C Pack_Body 87/03/12 12:40:11 Phil     7499  ;
    .Scanner                 : C Proc_Body 87/03/12 12:41:08 Phil     7283  ;
  Copyright_1986_Rational    :   Text      87/02/26 21:48:13 Phil      279  ;
  Find_Null_Acls             : C Proc_Spec 87/02/10 16:04:30 Phil     7460  ;
  Find_Null_Acls             : C Proc_Body 87/03/09 10:27:46 Phil    23826  ;
  Fix_Images                 : C Proc_Spec 87/02/10 16:04:38 Phil     7485  ;
  Fix_Images                 : C Proc_Body 87/03/12 12:38:48 Phil     7512  ;
  Restricted_Rights_Legend   :   Text      87/02/26 21:48:14 Phil      517  ;
  Setup_Acls                 : C Load_Proc 87/03/09 17:04:13 Phil     7304  ;
  Set_Universe_Acls          : C Proc_Spec 87/02/10 16:04:40 Phil     7223  ;
  Set_Universe_Acls          : C Main_Body 87/03/09 16:49:06 Phil    32525  ;
```

Many of the following examples illustrate the use of the Copy procedure. Note that a Copy is actually a Save followed by a Restore.

Table 4-1. Options Available by Procedure Names

Option	*Copy*	*List*	*Restore*	*Save*
After	X			X
Become_Owner	X		X	
Cdb	X		X	X
Default_Acl	X		X	
Format		X	X	X
Label		X	X	X
Nonrecursive	X	X	X	X
Object_Acl	X		X	
Overwrite	X		X	
Prefix				X
Primary	X		X	
Promote	X		X	
Replace	X		X	
Revert_Cdb	X		X	
World_Acl	X		X	
Version				X

Example 1

This example uses the Copy procedure to copy a world from one location to another on the same machine, repromote the Ada units within the world, and replace already existing objects with the same name as those being copied. The following command accomplishes this:

```
Archive.Copy (Objects => "!users.phil.tools",
              Use_Prefix => "!users.phil.test_area",
              For_Prefix => "*",
              Options => "promote,replace",
              Response => "<PROFILE>");
```

- The Objects parameter specifies the name of the world to be copied.
- The Use_Prefix parameter specifies the location for the copy of that world and its contents.
- The For_Prefix parameter ("*") causes the Use_Prefix name to replace the entire name specified in the Objects parameter. The Replace option is included because we may want to execute this Copy procedure more than once, replacing installed or coded units that were already in !Users.Phil.Test_Area. Without this option, the Copy procedure would not overwrite an Ada unit in !Users.Phil.Test_Area unless it were in the source state. The Promote option causes units to be restored to their original (in this case, coded) state.

Example 2

This example uses the Copy procedure to move updates made in !Users.Phil.Tools to !Users.Phil.Test_Area. The current context is !Users.Phil.Tools. Any units in !Users.Phil.Tools and not in !Users.Phil.Test_Area will be copied. In addition, any units that appear in both !Users.Phil.Tools and !Users.Phil.Test_Area that have a more recent update time in !Users.Phil.Tools will be copied. The following command accomplishes this:

```
Archive.Copy (Objects => "?",
              Use_Prefix => "!users.phil.test_area",
              For_Prefix => "$",
              Options => "overwrite=>updated_objects  replace",
              Response => "<PROFILE>");
```

- The Objects parameter specifies all objects enclosed by the current context recursively (including nested directories and subunits of Ada units).
- The For_Prefix parameter specifies that the name of the current context should be replaced by the Use_Prefix string.

Note that For_Prefix => "*" will not work in this case because Objects => "?" does not specify a unique prefix (it is a naming wildcard and matches objects in the current world).

Example 3

This example uses the Copy procedure to move the links and switch file associations from one world to another. Links and other world-associated information will be moved from the current context world (!Users.Phil.Tools) to the world !Users.Phil.Test_Area. The following command accomplishes this:

```
Archive.Copy (Objects => "$",
              Use_Prefix => "!users.phil.test_area",
              For_Prefix => "*",
              Options => "nonrecursive",
              Response => "<PROFILE>");
```

- The For_Prefix => "*" parameter is appropriate here because the Objects parameter specifies a unique object whose name will be changed to the Use_Prefix string during the copy.
- The Nonrecursive option causes only the objects named exactly by the Objects parameter to be moved, excluding any enclosed objects. Thus, only the world is copied. Moving the world causes the links and associations to be copied.

Example 4

This example uses the Copy procedure to copy units from !Users.Phil.Tools to !Users.Phil.Test_Area_2 and change their names so that each unit and file has the prefix Sim_. The following command accomplishes this:

```
Archive.Copy (Objects => "!users.phil.tools.@",
              Use_Prefix => "!users.phil.test_area_2.sim_@",
              For_Prefix => "!users.phil.tools.@",
              Options => "",
              Response => "<PROFILE>");
```

- The Objects parameter specifies just units in the Tools library. Since the world Tools itself is not to be renamed, it must be excluded from the objects to be saved. If it is necessary to copy links and switch associations for the world, this must be done in a separate command.
- The Use_Prefix and For_Prefix parameters specify the prefix string to be replaced and substituted. Note the use of the *at* sign (@) wildcard in the For_Prefix parameter and the *at* sign (@) substitution character in the Use_Prefix parameter.

Note that the Ada source for the renamed units will be changed to the new unit names. Other references to the old names within the units will not be changed, which may cause a unit to fail to promote.

The resulting world is:

```
!Users.Phil.Test_Area_2  : World;
  Sim_Cg_Switches              : S Pack_Spec 87/03/12 12:40:51 Phil    7263  ;
  Sim_Cg_Switches              : S Pack_Body 87/03/12 12:40:11 Phil    7499  ;
  Sim_Copyright_1986_Rational  :   Text      87/02/26 21:48:13 Phil     279  ;
  Sim_Find_Null_Acls           : S Proc_Spec 87/02/10 16:04:30 Phil    7460  ;
  Sim_Find_Null_Acls           : S Proc_Body 87/03/09 10:27:46 Phil   23826  ;
  Sim_Fix_Images               : S Proc_Spec 87/02/10 16:04:38 Phil    7485  ;
  Sim_Fix_Images               : S Proc_Body 87/03/12 12:38:48 Phil    7512  ;
  Sim_Restricted_Rights_Legen
                               :   Text      87/02/26 21:48:14 Phil     517  ;
  Sim_Setup_Acls               : C Load_Proc 87/03/09 17:04:13 Phil    7304  ;
  Sim_Set_Universe_Acls        : S Proc_Spec 87/02/10 16:04:40 Phil    7223  ;
  Sim_Set_Universe_Acls        : S Main_Body 87/03/09 16:49:06 Phil   32525  ;
```

Example 5

This example uses the Copy procedure to copy a set of objects. The following command accomplishes this:

```
Archive.Copy (Objects => "[fix_images'spec,copy@]",
              Use_Prefix => "!users.phil",
              For_Prefix => "$",
              Options => "",
              Response => "<PROFILE>");
```

- The Objects parameter uses naming set notation and a wildcard in the specification of the objects to copy.
- Note that the For_Prefix parameter must not be "*", because that would cause each object to be renamed "!Users.Phil."

Example 6

This example uses the Copy procedure to copy objects onto another machine, using Rational Networking—TCP/IP. On the new machine, the objects will retain the names they had on the old machine. The following command accomplishes this:

```
Archive.Copy (Objects => "!users.phil.test_area",
              Use_Prefix => "!!zebra",
              For_Prefix => "*",
              Options => "",
              Response => "<PROFILE>");
```

- The Objects parameter specifies the objects to be moved.
- The Use_Prefix parameter specifies the new machine name (Zebra) but not another directory. The objects will have the same name on Zebra as the source machine.

Note that links are created to point to same named units on Zebra.

Example 7

This example uses the Copy procedure to copy objects into another library on another machine, using Rational Networking—TCP/IP. The objects will be renamed on the new machine. The following command accomplishes this:

```
Archive.Copy (Objects => "!users.phil.test_area",
              Use_Prefix => "!!zebra!delta_project.tools",
              For_Prefix => "*",
              Options => "",
              Response => "<PROFILE>");
```

- The Use_Prefix parameter specifies both the machine name and the new library name.
- If libraries named in the Use_Prefix parameter need to be created, they are created as worlds. In this case, if Delta_Project did not exist, it would be created.

Example 8

This example uses the Save procedure to save objects as a backup. The example makes an archive save file of the contents of !Users.Phil.Tools. All contained objects will be included. The following command accomplishes this:

```
Archive.Save (Objects => "!users.phil.tools",
              Options => "R1000",
              Device => "!users.phil.backups.tools_87_03_09",
              Response => "<PROFILE>");
```

- The Objects parameter specifies the objects to be saved.
- The Device specifies the directory to be created to contain the save.

The Restore procedure can later be used to restore individual objects or all objects saved.

Example 9

This example uses the Save procedure to distribute a set of subsystems (and other units) to other machines.

The file Distribution_Contents contains:

```
!commands.internal.maintenance
!commands.internal.maintenance.[state?,  logs, compatibility??]
!commands.internal.maintenance.[rev9_0_spec?,  code9_0_0?]
!commands.internal.release_tools
!commands.internal.release_tools.[state?,  logs, compatibility??]
!commands.internal.maintenance.[rev9_0_spec?,  rev9_0_0?]
```

The following command accomplishes the distribution of the subsystems:

```
Archive.Save (Objects => "_distribution_contents",
              Options => "nonrecursive",
              Device => "!users.phil.release9_0_0",
              Response => "<PROFILE>");
```

- The Objects parameter specifies an indirect file. When a subsystem view is to be saved and restored, the containing subsystem must be present at the restore site (or restored as part of the restoration of the view). This is accomplished in this example by saving the containing subsystem, its state directory, and compatibility information. Failure to do this may yield a view that cannot be executed properly.
- The Nonrecursive option will cause only the objects specifically named in the file to be saved, not the contained objects.
- The Device parameter names the directory that will contain the save set. The default value for Device (not used in this example) causes saved objects to be written onto magnetic tape.

The result of the save is:

```
!Users.Phil.Release9_0_0  : World;
  Data  :   File      87/03/09 18:20:56 Phil    68386    ;
  Index :   Text      87/03/09 18:20:56 Phil    61408    ;
```

Examples 10 through 12 illustrate different ways of restoring the objects saved in this example.

Example 10

This example uses the Restore procedure to partially restore the units saved in Example 9 into a renamed subsystem view. The following command accomplishes this:

```
Archive.Restore
       (Objects => "!commands.internal.maintenance.rev9_0_spec",
        Use_Prefix => "!commands.internal.maintenance.rev9_1_spec"
        For_Prefix => "!commands.internal.maintenance.rev9_0_spec",
        Options =>
        Device => "!users.phil.release9_0_0",
        Response => "<PROFILE>");
```

- The Objects parameter specifies the name of the objects to restore from the save set. In this case, it specifies a spec view name, causing the spec view and its contents to be restored.
- The Use_Prefix and For_Prefix parameters cause the restored view to be given a new name. Note that For_Prefix => "*" cannot be used in this example because the default that would be used for the For_Prefix parameter is the Objects parameter of the Save, not the Restore procedure, and the Save procedure does not specify a unique prefix to be replaced by the Use_Prefix parameter string.
- The Device parameter specifies the source for the restore; in this example, it was a directory that was the destination for the Archive.Save.

Example 11

This example uses the Restore procedure to restore updated objects saved in Example 9. All of the restored units will be promoted. The following command accomplishes this:

```
Archive.Restore (Objects => "?",
                 Use_Prefix => "*",
                 For_Prefix => "*",
                 Options => "promote, overwrite=updated",
                 Device => "!users.phil.release9_0_0",
                 Response => "<PROFILE>");
```

- The Objects parameter specifies that all objects from the save are to be restored.
- The Use_Prefix and the For_Prefix parameters specify that objects are to be restored under their original names as saved.
- The Promote option causes Ada units to be promoted to the unit state they were in when they were saved.
- The Overwrite=Updated option causes existing objects with the same names as those from the save to be left alone if the update time from the restore is earlier than the last update time of the object.

Example 12

This example uses the Copy procedure to update changed objects saved in Example 9. This will allow changes made in !Commands.Internal.Maintenance.Rev9_0_Spec to be moved from this machine to another machine named Shelby. The following command accomplishes this:

```
Archive.Copy
   (Objects => "!commands.internal.maintenance.rev9_0_spec",
    Use_Prefix => "!!shelby",
    For_Prefix => "*",
    Options => "after=03/01/87 replace overwrite=changed_objects promote",
    Response => "<PROFILE>");
```

- The After option specifies that only objects modified after 3/1/87 are to be considered.
- The Replace option allows Ada units in other than the source state to be overwritten.
- The Overwrite option causes only objects on Shelby that changed to be moved.
- The Promote option causes moved objects to be promoted to their original unit state.
- Compatibility information needed for any objects is automatically copied as well.

Example 13

This example uses the Copy procedure to update the compatibility database for a secondary for incremental insertions.

Suppose a change has been made in !Commands.Internal.Maintenance.Rev9_0_Spec. This change will be moved to a secondary subsystem on another machine called Shelby. Rather than copying the entire spec (which requires demoting its closure), the addition can be made incrementally by inserting the new declarations into the coded spec on Shelby (the secondary).

To accomplish this, the compatibility information for the subsystem must first be moved from the primary to the secondary. Then the incremental change can be made compatibly to the secondary.

The following command moves the compatibility information:

```
Archive.Copy (Objects => "!commands.internal.maintenance",
              Use_Prefix => "!!shelby",
              For_Prefix => "*",
              Options => "compatibility_database",
              Response => "<PROFILE>");
```

Now the incremental changes can be made and compatibility maintained.

Hints for Using the Procedures in Package Archive

- Do not mix set notation (for example, [a,b,c]) with substitution characters in the Use_Prefix and For_Prefix parameters.
- Use caution when trying to move several objects by specifying the Use_Prefix and For_Prefix parameters. You must ensure that they will match.
- Be careful when archiving units out of a subsystem units directory. You must reassociate the switch file to remove connections to the subsystem.
- If you are restoring a wide range of objects in a number of directories to which you may not have access, run the Restore procedure in privileged mode.
- If you are moving a primary subsystem to a new location, rather than simply making a release copy of a subsystem, you need to specify the Primary option on the Copy or Restore procedure. Also remember to change the original subsystem to a secondary if it is not being deleted. This can be done with the !Compiler_Interface.Compatibility.Make_Secondary command.
- When moving large numbers of objects between machines, rather than using the Copy procedure, use Save and Restore from tape or the following procedure. Use Save with the Device parameter specifying a library. Copy that library onto the other machine. Then use Restore with the Device parameter, specifying the copy of the library. This eliminates a large amount of network traffic and improves efficiency. However, more disk space is required to build the library for archives.

Error Response

The commands in this package have a Response parameter that specifies how the command should respond to errors, how to generate logs, and what activities to use. The response profile "<PROFILE>", which many commands use by default, specifies the job response profile. If there is no job response profile, the session response profile ("<SESSION_PROFILE>") is used. If there is no session response profile, the system's default profile ("<DEFAULT>") is used. For further information on profiles, see SJM, package Profile.

procedure Copy

```
procedure Copy (Objects     : String := "<IMAGE>";
                Use_Prefix  : String := "*";
                For_Prefix  : String := "*";
                Options     : String := "";
                Response    : String := "<PROFILE>");
```

Description

Copies one or more objects from one location to another, either on the same R1000 or between two R1000s that are connected through Rational Networking—TCP/IP.

The Copy procedure leaves the original source objects in place, creating a copy in the location indicated by the Use_Prefix parameter. The Copy procedure also:

- Reconstructs the saved objects in their original hierarchical structure.
- Optionally promotes Ada units to the states they were in when they were copied.
- Changes the location of objects as specified in the Use_Prefix and For_Prefix parameters.
- Rebuilds the links for copied worlds.
- Reassociates copied libraries with their switch files.
- Permits the user to change the ACLs and default ACLs associated with the copied worlds and objects.

By default, the Copy procedure copies all of the objects named by the Objects parameter, including their sublibraries and subunits. Also by default, when copied, objects will overwrite existing Environment objects of the same name, unless the existing objects either are frozen (see the Library.Freeze procedure) or are installed or coded and have dependents. The overwriting of objects can be controlled further by specifying an Overwrite option as part of the Options parameter.

Note that objects can alternatively be moved between R1000s by saving objects onto tape with the Save procedure on one R1000 and then using the Restore procedure on the other R1000 to read the tape. This method is recommended for moving a large number of objects. For moving fewer objects, or for copying objects to a different location on the same R1000, the Copy procedure is a convenient, one-step alternative that does not require a tape or an intermediate library, but it does require Rational Networking—TCP/IP.

Parameters

```
Objects : String := "<IMAGE>";
```

Specifies the object or objects to be copied. The parameter takes any naming expression. By default, the current image is copied, unless there is a selection in that image, in which case the selected object is copied. By using the defaults, the specified objects and all contained objects are copied.

The Objects parameter may begin with an optional machine name followed directly by the fully qualified pathnames of one or more objects. For example:

```
objects=>"!!ml!users.anderson.statistics"
```

The machine name has the form !!*name*, where *name* names the R1000 on which the source objects reside. (The name of an R1000 is typically displayed in the banner under the Message window.) You can omit the machine name if you are entering the command on the R1000 that contains the objects to be copied. See the *Rational Networking—TCP/IP Reference Manual* for further information.

Both the Use_Prefix and Objects parameters can specify a machine name, which allows copying of data between two machines to be initiated from yet a third machine.

You can copy objects such as worlds, directories, Ada units, text files, data files, switch files, activities, subsystems, and subsystem views.

You can specify more than one object by using wildcards, context characters, special names, set notation, or an indirect file. For more information about indirect files and naming, see the Key Concepts in this book.

```
Use_Prefix :  String := "*";
```

Specifies where to rebuild the copied objects by allowing the user to change the names of copied objects when they are copied. If the Use_Prefix parameter has the special default value "*", the objects are copied using the original name. This is useful only when copying objects from one R1000 into the identical library structure on another R1000. The "*" prefix cannot be used when copying objects on the same machine.

If the Use_Prefix parameter is not the special default value "*", it must specify the name of an object into which the objects can be copied. The Use_Prefix string consists of an optional machine name followed directly by an optional fully qualified object name. For example:

```
use_prefix=>"!!m2!project_1"
```

specifies that objects are to be copied into library !Project_1 on machine M2.

A machine name has the form !!*name*, where *name* specifies the R1000 onto which the source objects are copied. You can omit the machine name if you are entering the command on the destination R1000. If neither the Objects parameter nor the Use_Prefix parameter specifies a machine name, the source objects are copied from the source to the location specified by the Use_Prefix parameter on the current machine.

Both the Use_Prefix and Objects parameters can specify a machine name, which allows copying of data between two machines to be initiated from yet a third machine.

The name for a copied object is derived from the name of the original object by replacing the shortest portion matched by the For_Prefix parameter with the value of the Use_Prefix parameter. For example:

```
Copy (objects=>"!!m3!users.hjl.cli.cmd",
      use_prefix=>"!users.bar",
      for_prefix=>"!users.hjl.cli");
```

copies the file !Users.Hjl.Cli.Cmd on machine M3 to !Users.Bar.Cmd on the machine where the command is executed. If !Users.Bar does not exist, it is created as a new world.

If the For_Prefix parameter does not match an object, that object is copied under its original name.

The For_Prefix parameter can use wildcard characters and the Use_Prefix parameter can use substitution characters. If wildcards are present in the For_Prefix parameter, the Use_Prefix parameter must specify, through the use of substitution characters, the full name of the target object. For further information, see the Key Concepts in this book.

Table 4-2. Making Copies with For_Prefix and Use_Prefix

For_Prefix	*Use_Prefix*	*Resulting Object*
!A.B	!X	!X.C.D
[!A,!A.B]	!X	!X.B.C.D
[!R.S,!A.B]	!X	!X.C.D

```
For_Prefix :  String := "*";
```

Participates in the specification of the location for rebuilding the copied objects by allowing the user to change the names of the objects when they are copied.

When the For_Prefix parameter has the default value "*", the copied objects are copied using the prefix that is stored with them. The For_Prefix parameter is ignored when the Use_Prefix has the special value "*".

When the value of the Use_Prefix parameter is not "*", the shortest portion of each object that is matched by the For_Prefix parameter is replaced by the Use_Prefix parameter. If wildcards are present in the For_Prefix parameter, the Use_Prefix parameter must specify, through the use of substitution characters, the full name of the target object. In this way, copied objects can be rebuilt under different names.

An object that is not matched by the For_Prefix parameter is not copied and a warning message is generated.

When the Objects parameter names multiple objects, the For_Prefix parameter should specify a string common to all of their names. For example:

```
copy (objects=>"!!m3!users.hjl.cli.cmd",
      use_prefix=>"!users.bar",
      for_prefix=>"!users.hjl.cli");
```

copies the file !Users.Hjl.Cli.Cmd on machine M3 to !Users.Bar.Cmd on the same machine. If !Users.Bar does not exist, it is created as a new world.

The For_Prefix parameter can use wildcard characters, and the Use_Prefix parameter can use substitution characters. For further information, see the Key Concepts in this book.

Table 4-2 illustrates the result of copies made with representative combinations of the Use_Prefix and For_Prefix parameters. The object stored is !A.B.C.D.

```
Options :  String := "";
```

Specifies options to be used in copying objects. The Options parameter allows you to specify more specifically which objects to copy, the compilation state of the copied objects, and the access lists (ACLs) for the copied objects. You can specify more than one option. For further information on specifying options in Options parameters, see the Key Concepts in this book.

After=*time*

Copies only objects that have been changed more recently than the specified *time*, which can be a date, a time of day, or both, and can be written in any of the styles defined by the !Tools.Time_Utilities.Date_Format type and the !Tools.Time_Utilities.Time_Format type. For example, specifying the following value for the Options parameter copies only those objects named by the Objects parameter that were updated after June 11, 1987, at noon:

```
options=>"after=(06/11/87  12:00)"
```

Become_Owner

Specifies that the access lists (ACLs) of all copied worlds should be modified to give the copying username owner access to the copied worlds.

If a group specified in the ACL for an object does not exist on the machine onto which the object is copied, the ACL entry for that group is removed from the object's ACL.

Compatibility_Database=*subsystems*

Specifies that the full compatibility database for each subsystem specified should be copied. When Ada units in a subsystem are copied, the relevant portions of the subsystem compatibility database are automatically copied with them. Therefore, this option is required only in special situations, primarily when a primary and a secondary subsystem need to be synchronized. This option can be specified as CDB, if desired. For further information on subsystems and compatibility databases, see Project Management (PM).

If no subsystems are specified in the *subsystems* portion of this option, the Objects parameter specification is used instead. Any subsystem views specified by the Objects parameter will have the full compatibility information saved for them.

The Nonrecursive option does not affect the interpretation of the CDB specification.

To copy compatibility databases only, use:

```
archive.copy (objects=>"subsystems",
              options=>"cdb");
```

where *subsystems* is the name of the subsystems whose CDBs are to be copied. This will generate an error if the Objects parameter specifies nonsubsystems.

To copy compatibility databases with other objects, use:

```
archive.copy (objects=>"other",
              options=>"cdb=subsystems");
```

where *other* specifies objects that are probably disjoint from *subsystems*.

Default_Acl=*new acl*

Specifies a new default access list (ACL) for copied worlds. The default ACL is used for new objects created within a world after the copy is complete. The default ACL copied with the world will be changed to the one specified, which can be a new ACL or the special values Inherit or Archived. Inherit means to

use the standard inheritance rules for new versions of objects. Archived means to use the ACL archived with the object, which is the default for the Default_Acl option. New ACLs must follow the syntax rules for ACLs (for further information, see "Access Control" in the Key Concepts in this book).

If a group specified in the ACL for an object does not exist on the machine onto which the object is copied, the ACL entry for that group is removed from the object's ACL.

`Nonrecursive` Boolean

Specifies that only the objects that are actually named by the Objects parameter should be copied. Sublibraries and subunits within the named objects are not copied, unless explicitly specified. This option allows you to copy the links and switches associated with libraries without copying the objects in those libraries. For example, it allows you to copy a library plus a subset of its contents without copying the entire library.

This option prevents the subcomponents of libraries and Ada units from being implicitly copied. For example:

```
archive.copy (objects=>"[!users.hjl,!users.hjl.cli,!users.hjl.cli.@]",
              options=>"nonrecursive");
```

copies only the named objects and not their substructures.

`Object_ACL=`***new acl***

Specifies a new ACL for copied objects. The ACL copied with the objects will be changed to the one specified, which can be a new ACL, or the special values Inherit or Archived. Inherit means to use the standard inheritance rules for new versions of objects. Archived means to use the ACL copied with the object, which is the default. New ACLs must follow the syntax rules for ACLs (for further information, see "Access Control" in the Key Concepts in this book).

If a group specified in the ACL for an object does not exist on the machine onto which the object is copied, the ACL entry for that group is removed from the object's ACL.

`Overwrite` literal

Allows you to control overwriting further by either allowing or preventing the transfer of objects that can overwrite existing objects. The Overwrite option works in conjunction with the Replace option: Overwrite selects objects that are eligible to be copied, and Replace then determines what happens to the existing target objects.

`All_Objects`	Use the Overwrite=All_Objects option (the default) to cause all objects to be copied, overwriting any existing objects to the extent permitted by the Replace option.
`New_Objects`	Use the Overwrite=New_Objects option to prevent the copying of any archived object that will overwrite an existing object. That is, this option allows objects to be rebuilt only under new names or in new locations and prevents existing objects from being overwritten.

Updated_Objects	Use the Overwrite=Updated_Objects option to copy only those archived objects that were modified more recently than the corresponding existing objects. The more recent objects overwrite the existing objects. This option is especially useful when there is parallel development on multiple R1000s and you need to copy updated objects from one R1000 to another.
Changed_Objects	Use the Overwrite=Changed_Objects option to copy new and updated objects. This is equivalent to using both the New_Objects and Updated_Objects options, which are not illegal when used together in an Options list.

Primary Boolean

Specifies, when true, that the compatibility database should be copied as a primary (the CDB will have read/write access). When the value is false (the default), the compatibility database will be copied as a secondary (read access only).

For further information on subsystems and compatibility databases, see Project Management (PM).

Promote Boolean

Causes, when true, the Copy procedure to attempt to promote copied Ada units to the states (installed or coded) in which they were archived. When this option is not used, all Ada units are rebuilt in the source state.

Note that, when you copy a world, the links for that world are copied. However, links can be rebuilt only if the units to which the links refer have the same source name. If you are copying the world onto a different R1000 that does not contain those units, then the links cannot be rebuilt. Objects that depend on the missing units cannot be promoted.

When a link or switch cannot be resolved, the system checks in a file called !Machine.Archive_Mappings. This file allows you to rename the sources of links and the names of switches during the restore process. To use this facility, you must create a text file called !Machine.Archive_Mappings, which contains the old name and the new name. When a link fails to be added or a switch fails to be set, the !Machine.Archive_Mappings file is searched for an occurrence of the old name. If it is found, the operation is repeated using the new name.

The Archive_Mappings file has the following format:

old link name

new link name

...

(repeat in pairs for all links) ...

old switch name

new switch name

...

(repeat in pairs for all switches)

...

Replace

Given that an object that is being copied already exists on the target, this option works in conjunction with the Overwrite option to cause the Copy procedure to do the following:

- If the target object is frozen, it will be unfrozen.
- If the target object is an installed or coded Ada unit with clients, it is demoted to source using the Compilation.Demote procedure with "<ALL_WORLDS>" specified in the Limit parameter.
- If the parent library into which an object is being restored is frozen, the parent will be unfrozen to restore the object. After the object is restored, it will be frozen again.

Note that the Replace option does not override access control.

Revert_Cdb Boolean

Specifies, when true, that a less recently updated version of a compatibility database can be copied over a more recently updated version. When the value is false (the default), the copy will fail if the compatibility database is less recent than the one currently on the machine.

For further information on subsystems and compatibility databases, see Project Management (PM).

World_Acl=*new acl*

Specifies a new access list (ACL) for restored worlds. The ACL copied with the world will be changed to the one specified, which can be a new ACL or the special values Inherit or Archived. Inherit means to use the standard inheritance rules for new versions of objects. Archived means to use the ACL copied with the object, which is the default. New ACLs must follow the syntax rules for ACLs (for further information, see "Access Control" in the Key Concepts in this book).

If a group specified in the ACL for an object does not exist on the machine onto which the object is copied, the ACL entry for that group is removed from the object's ACL.

Response : String := "<PROFILE>";

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

Example 1

The following command copies a world from the current R1000 onto an R1000 named M2:

```
archive.copy (objects=>"!users.anderson",
              use_prefix=>"!!m2");
```

The world keeps its original name. The result is that the world !Users.Anderson will exist on machine M2. If the copying username has operator capability, a new user Anderson will be created on M2 with the same password. The contents of the world are also restored.

Example 2

The following command copies a world from an R1000 named M2 onto an R1000 called M1:

```
archive.copy (objects=>"!!m2!users.doyle",
              use_prefix=>"!!m1",
              for_prefix=>"*");
```

The world keeps its original name. If the initiator of the command has operator capability, the system will attempt to create a user with the same name. In this example, the contents of !Users.Doyle are copied onto M1. If the copying username has operator capability, a new user Doyle is created.

Example 3

The following command copies two directories onto an R1000 named M2. Because the Use_Prefix parameter does not have the default value, these directories are not rebuilt in !Users.Anderson but are put in a parent library called !Project_1, which will be created as a world if it does not already exist.

```
archive.copy (objects=>"!users.anderson[tools, utils]",
              use_prefix=>"!!m2!project_1",
              for_prefix=>"!users.anderson");
```

As a result of executing this command, machine M2 contains directories !Project_1-.Tools and !Project_1.Utils.

See the introduction to this package for more examples.

---

procedure List

```
procedure List (Objects   : String := "?";
                Options   : String := "R1000";
                Device    : String := "MACHINE.DEVICES.TAPE_0";
                Response  : String := "<PROFILE>");
```

Description

Produces a listing of the objects that were saved on the specified tape or library.

By default, the listing is put in the Current_Output file, which typically appears as a window on your screen.

Parameters

`Objects : String := "?";`

Specifies the objects that should appear in the listing. The default value (?) is a pattern that lists everything.

Patterns include the characters #, @, ?, [], and ~, as described below:

- # The pound sign is replaced by a single character. It will not match the null string or a period (.). For example, File_# matches File_1 and File_2. The pound sign does not match the null string.
- @ The *at* sign matches any string that does not contain a period. For example, !Users.Mary.@ matches everything in Mary's home world, but nothing below that level in the library. The *at* sign matches the null string.
- ? The question mark matches any sequence of characters at the beginning of the name (that is, ? or !?) or a sequence of characters beginning with the period (.). It matches the null string. For example, !Users? matches everything in !Users.
- [] The brackets indicate sets of objects—for example, [!Users.Mary?, !Users.John?].
- ~ The tilde indicates that something should not be restored—for example, ~!Users.Mary indicates that !Users.Mary should not be restored.

A nondefault value must be a pattern, a fully qualified object name starting with the ! prefix (or a pattern that matches !), or a set of fully qualified object names enclosed in brackets.

`Options : String := "R1000";`

Specifies options to be used in listing objects. You can specify more than one option. For further information on specifying options, see the Key Concepts in this book.

Format literal

Specifies one of three formats for the Index and Data files that were saved. Format is irrelevant if Device is a library. If Device is a tape, Format must specify one of the following values and must be the same as the tape that was saved.

R1000 — Use the Format=R1000 option if the save was made with this option. This format produces a single volume set formatted in chained ANSI:

- The volume set *DATA_SHORT* consists of a single tape that contains the Data file followed by the Index file. The images of the objects being saved are written directly onto tape.

R1000_Long — Use the Format=R1000_Long option if the save operation specified this option, which would have required more than one tape. This format produces two volume sets, both formatted in chained ANSI:

- The volume set *DATA_LONG* consists of multiple tapes that contain the Data file.
- The volume set *INDEX* consists of a single tape that contains the Index file.

Format=Ansi — Use the Format=ANSI option if the list is being performed to list a tape saved for or from a non-R1000 machine that accepts ANSI tapes. If you require this capability, see your Rational technical representative. This option writes the data into a temporary file and then writes both the Index and the Data files onto a tape using ANSI tape facilities.

- The volume set has no name and consists of a single tape that contains the Data file followed by the Index file.

Label=*string*

Specifies the string, used for identification purposes, that is written onto the tape at the head of the archived data, where *string* is any user-specified string. This label is not the same as the volume identifier or the volume set name. Typically, it is used to record the tape owner's name, the date, and so on. The List procedure can use this option to verify the label on the tape. The verification is not case-sensitive. If the label doesn't match, the tape is rejected as if it were the wrong volume.

The Label can be a single text line of arbitrary length. A *string* that contains special characters (commas or semicolons) must use balanced sets of parentheses to indicate that it is a string and thus should not be interpreted as special characters. For example, an option containing the comma (,) characters can be specified as "LABEL=(MONDAY, JUNE 1, 1987)".

For further information on specifying strings in an Options parameter, see the Key Concepts in this book.

Nonrecursive Boolean

Specifies that only the objects that are actually named by the Objects parameter should be listed. This option does not recursively list sublibraries and subunits within the named objects. It also allows you to list a library plus a subset of its contents without listing the entire library.

```
Device : String := "MACHINE.DEVICES.TAPE_0";
```

Specifies the name of the device from which the archived objects are to be accessed. The default value for Device reads from tape. If the objects are saved in a library rather than on tape, you can supply a library name.

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

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

References

procedure Save

SMU, package Tape

---

procedure Restore

```
procedure Restore (Objects     : String := "?";
                   Use_Prefix  : String := "*";
                   For_Prefix  : String := "*";
                   Options     : String := "R1000";
                   Device      : String := "MACHINE.DEVICES.TAPE_0";
                   Response    : String := "<PROFILE>");
```

Description

Reads a tape or library that was created by the Save procedure and restores the specified object or objects.

The tape or library is specified by the Device parameter.

The Restore procedure also:

- Reconstructs the structure of the saved objects.
- Optionally promotes Ada units to the states they were in when they were saved.
- Rebuilds objects either to their original location in the Environment or to the location specified in the command.
- Rebuilds the links for restored worlds.
- Reassociates restored directories with their switch files.
- Permits the restorer to change the default access list (ACL) or ACL associated with worlds and objects within those worlds.

Note that you can use the Restore procedure to rebuild a subset of the objects saved on a tape or in a library.

By default, when archived objects are restored, they will overwrite existing Environment objects of the same name, unless the existing objects either are frozen (see the Library.Freeze procedure) or are installed and have dependents.

Parameters

```
Objects : String := "?";
```

Specifies the object or objects to be restored from the tape or library. The default value (?) is a pattern that restores everything.

Patterns include the characters #, @, ?, [], and ~, as described below:

- # The pound sign is replaced by a single character. It will not match the null string or a period (.). For example, File_# matches File_1 and File_2. The pound sign does not match the null string.
- @ The *at* sign matches any string that does not contain a period. For example, !Users.Mary.@ matches everything in Mary's home world, but nothing below that level in the library structure. The *at* sign matches the null string.
- ? The question mark matches any sequence of characters at the beginning of the name (that is, ? or !?) or a sequence of characters beginning with the period (.). It matches the null string. For example, !Users? matches everything in !Users.
- [] The brackets indicate sets of objects—for example, [!Users.Mary?, !Users.John?].
- ~ The tilde indicates that something should not be restored—for example, ~!Users.Mary indicates that !Users.Mary should not be restored.

The value of this parameter must be a pattern, a fully qualified pathname starting with the ! prefix (or a pattern that matches "!"), or a set of fully qualified pathnames enclosed in brackets. For example:

```
Objects => "!Machine.Error_Logs.Log_86_04_@"

Objects => "[!Users.Doyle.Tools, !Project_1.Utils]"
```

```
Use_Prefix :  String := "*";
```

Specifies where to rebuild the restored objects by allowing the user to change the names of archived objects when they are restored. If the Use_Prefix parameter has the special default value "*", the objects are restored using the original name. This is useful when restoring objects into identical library structures between two machines.

When the value of the Use_Prefix parameter is not the special default value "*", it must specify the name of an object into which the archived objects can be restored. The name for a restored object is derived from the name of the archived object by replacing the shortest portion matched by the For_Prefix parameter with the value of the Use_Prefix parameter. For example:

```
archive.restore (objects=>"!users.hjl.cli.cmd",
                 use_prefix=>"!users.bar",
                 for_prefix=>"!users.hjl.cli");
```

restores the file !Users.Hjl.Cli.Cmd to !Users.Bar.Cmd on the current machine.

The For_Prefix parameter can use wildcard characters and the Use_Prefix parameter can use substitution characters. If wildcards are present in the For_Prefix parameter, the Use_Prefix parameter must specify, through the use of substitution characters, the full name of the target object. For further information, see the Key Concepts in this book.

```
For_Prefix :  String := "*";
```

Participates in the specification of the location for rebuilding the restored objects by allowing the user to change the names of the archived objects when they are restored.

When the For_Prefix parameter is the default value "*", the restored objects are copied using the prefix stored with the archived data using the Prefix option. The For_Prefix parameter is ignored when Use_Prefix has the special value "*".

When the value of Use_Prefix is not the special default "*", the shortest portion of the For_Prefix parameter is replaced by the Use_Prefix parameter. If wildcards are present in the For_Prefix parameter, the Use_Prefix parameter must specify, through the use of substitution characters, the full name of the target object. In this way, restored objects can be rebuilt under different names.

An object that is not matched by the For_Prefix parameter is not copied, and a warning message is generated.

Table 4–3 illustrates the results of restoring with representative combinations of For_Prefix and Use_Prefix. The object stored on tape is !A.B.C.D.

Table 4–3. Restoring with For_Prefix and Use_Prefix

For_Prefix	*Use_Prefix*	*Resulting Object*
!A.B	!X	!X.C.D
[!A,!A.B]	!X	!X.B.C.D
[!R.S,!A.B]	!X	!X.C.D

```
Options :  String := "R1000";
```

Specifies options to be used in restoring objects. The Options parameter allows you to specify more specifically which objects to restore, the state they should be in when restored, and the access lists (ACLs) for the restored objects. You can specify more than one option. For further information on specifying Options, see "The Options Parameter" in the Key Concepts in this book.

Become_Owner

Specifies that the ACL of all restored worlds should be modified to give the restorer owner access to the restored worlds.

If a group specified in the ACL for an object when it is saved does not exist on the machine on which it is restored, the ACL entry for that group is removed from the object's ACL.

Compatibility_Database=*subsystems*

Specifies that the full compatibility database for each specified subsystem should be restored. When Ada units in a subsystem are archived, the relevant portions of the subsystem compatibility database are automatically archived with them. Therefore, this option is required only in special situations, primarily when a primary and a secondary subsystem need to be synchronized. This option name can be specified as CDB, if desired. For further information on subsystems and compatibility databases, see Project Management (PM).

If no subsystems are specified in the *subsystems* portion of this option, the Objects parameter specification is used instead. Any subsystem views specified by the Objects parameter will have the full compatibility database information restored for them.

The Nonrecursive option does not affect the interpretation of the CDB specification.

To restore compatibility databases only, use:

```
archive.restore (objects=>"subsystems",
                 options=>"cdb");
```

where *subsystems* is the name of the subsystems whose CDBs are to be restored. This will generate an error if the Objects parameter specifies nonsubsystems.

To restore compatibility databases with other objects, use:

```
archive.restore (objects=>"other",
                 options=>"cdb=subsystems");
```

where *other* is objects that are probably disjoint from *subsystems.*

Default_ACL=*new acl*

Specifies a new default ACL for restored worlds after the restore has completed. The default ACL is used for new objects created within a world after the restore is complete. The default ACL archived with the world will be changed to the one specified, which can be a new ACL or the special values Inherit or Archived. Inherit means to use the standard inheritance rules for new versions of objects. Archived means to use the ACL archived with the object, which is the default.

New ACLs must follow the syntax rules for ACLs (for further information, see "Access Control" in the Key Concepts in this book).

If a group specified in the ACL for an object when it is saved does not exist on the machine on which it is restored, the ACL entry for that group is removed from the object's ACL.

Format literal

Specifies one of three formats for the Index and Data files in which the archive was done. Format is irrelevant if Device is a library. If Device is a tape, the Format option must specify the same format used in the Save (that is, Format=R1000, Format=R1000_Long or Format=ANSI). The Format option is optional.

R1000 — Use the Format=R1000 option if the save operation required only one tape. This format produces a single volume set formatted in chained ANSI:

- The volume set *DATA_SHORT* consists of a single tape that contains the Data file followed by the Index file. The images of the objects that were saved were written directly onto tape.

R1000_Long — Use the Format=R1000_Long option if the save operation required more than one tape. This format produces two volume sets, both formatted in chained ANSI:

- The volume set *DATA_LONG* consists of multiple tapes that contain the Data file.
- The volume set *INDEX* consists of a single tape that contains the Index file.

Format=Ansi — Use the Format=ANSI option if the restore operation is being performed to restore a tape from a non-R1000 machine. If you require this capability, see your Rational technical representative.

- This option writes the data into a temporary file and then writes both the Index and the Data files onto a tape using ANSI tape facilities.
- The volume set has no name and consists of a single tape that contains the Data file followed by the Index file.

Label=*string*

Specifies the string, used for identification purposes, that is written onto the tape at the head of the archived data, where *string* is any user-specified string. This label is not the same as the volume identifier or the volume set name. Typically, it is used to record the tape owner's name, the date, and so on. The Restore procedure can be requested to verify the label on the tape. The verification is not case-sensitive. If the label does not match, the tape is rejected as if it were the wrong volume.

The label can be a single text line of arbitrary length. A *string* that contains special characters (commas or semicolons) must use balanced sets of parentheses to indicate that it is a string and thus should not be interpreted as special characters. For example, an option containing the special comma (,) character could be specified as "LABEL=(MONDAY, JUNE 1, 1987)".

For further information on specifying strings in an Options parameter, see "The Options Parameter" in the Key Concepts in this book.

Nonrecursive Boolean

Specifies that only the objects that are actually named by the Objects parameter should be restored. This option does not restore sublibraries and subunits within the named objects unless they are explicitly named. The option allows you to restore the links and switches associated with libraries without restoring the objects in those libraries. For example, with this option you can restore a library plus a subset of its contents without restoring the entire library.

This option prevents the subcomponents of libraries and Ada units from being implicitly restored. For example:

```
archive.restore (objects=>"[!users.hjl,!users.hjl.cli,!users.hjl.cli.@]",
                 options=>"R1000 nonrecursive");
```

will restore only the named objects and not their substructures.

Object_Acl=*new acl*

Specifies a new ACL for restored objects. The ACL restored with the object will be changed to the one specified, which can be a new ACL or the special values Inherit or Archived. Inherit means to use the standard inheritance rules for new versions of objects. Archived means to use the ACL archived with the object, which is the default. New ACLs must follow the syntax rules for ACLs (for further information, see "Access Control" in the Key Concepts in this book).

If a group specified in the ACL for an object when it is saved does not exist on the machine on which it is restored, the ACL entry for that group is removed from the object's ACL.

Overwrite literal

Allows you to control overwriting further by either allowing or preventing the transfer of objects that can overwrite existing objects. The Overwrite option works in conjunction with the Replace option. Overwrite selects objects that are eligible to be restored, and Replace then determines what happens to the existing target objects.

All_Objects	Use the Overwrite=All_Objects option to cause all objects to be restored, overwriting any existing objects to the extent permitted by the Replace option. This is the default.
New_Objects	Use the Overwrite=New_Objects option to prevent the restoring of any archived object that will overwrite an existing object. That is, this option allows objects to be rebuilt only under new names or in new locations and prevents existing objects from being overwritten.
Updated_Objects	Use the Overwrite=Updated_Objects option to restore only those archived objects that were modified more recently than the corresponding existing objects. The more recent objects overwrite the existing objects. This option is especially useful when there is parallel development on multiple R1000s and you need to restore updated objects from one R1000 to another.

Changed_Objects — Use the Overwrite=Changed_Objects option to restore new and updated objects. This is equivalent to using both the New_Objects and the Updated_Objects options, which are illegal when used together in an Options list.

Primary Boolean

Specifies whether to move the primary subsystem to a new location (see the introduction to this package for further information on primary and secondary subsystems). When true, the option specifies that the compatibility database should be restored as a primary (with read/write access). When false (the default), it specifies that the compatibility database should be restored as a secondary (with read access).

For further information on subsystems and compatibility databases, see Project Management (PM).

Promote

Causes, when used, the Restore procedure to attempt to promote restored Ada objects to the states (installed or coded) in which they were archived. When this option is not used, all Ada units are rebuilt in the source state.

Note that, when you restore a world, the links for that world are restored. However, links can be rebuilt only if the units to which the links refer exist. If you are restoring the world to a different R1000 that does not contain those units, then the links cannot be rebuilt. Objects that depend on the missing units cannot be promoted.

Replace

Given that an object that is being restored already exists on the target, this option works in conjunction with the Overwrite option to cause the Restore procedure to do the following:

- If the target object is frozen, it will be unfrozen.
- If the target object is an installed or coded Ada unit with clients, it is demoted to source using the Compilation.Demote procedure with "<ALL_WORLDS>" specified in the Limit parameter.
- If the parent library into which an object is being restored is frozen, the parent will be unfrozen to restore the object. After the object is restored, it will be frozen again.

Note that the Replace option does not override access control.

Revert_Cdb Boolean

Specifies, when true, that a less recently updated version of a compatibility database can be restored over a more recently updated version. When the value is false (the default), the restore of the compatibility database will fail if the compatibility database is less recent than the one currently on the machine.

For further information on subsystems and compatibility databases, see Project Management (PM) and the introduction to this package.

World_Acl=*new acl*

Specifies a new access list (ACL) for restored worlds. The ACL archived with the world will be changed to the one specified, which can be a new ACL or the special values Inherit or Archived. Inherit means to use the standard inheritance rules for new versions of objects. Archived means to use the ACL archived with the object, which is the default. New ACLs must follow the syntax rules for ACLs (for further information, see "Access Control" in the Key Concepts in this book).

If a group specified in the ACL for an object when it is saved does not exist on the machine on which it is restored, the ACL entry for that group is removed from the object's ACL.

```
Device : String := "MACHINE.DEVICES.TAPE_Ø";
```

Specifies the name of the device from which the archived objects are to be read. If you use the default value for Device, the Restore procedure reads from tape. If you supply a library name, the objects are read from the Data and Index files in that library.

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

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

Example 1

The following command restores !Project_1.Utils.Counter to its original location, because both the For_Prefix and the Use_Prefix parameters have default values:

```
archive.restore (objects=>"!project_1.utils.counter");
```

Example 2

The following command restores two worlds from tape and rebuilds them in the library !Users.Anderson:

```
archive.restore (objects=>"[!users.doyle.tools,!project_1.utils]",
                 use_prefix=>"!users.anderson",
                 for_prefix=>"[!users.doyle,!project_1]");
```

Because the Use_Prefix of "!Users.Anderson" is used, the restored objects are rebuilt as !Users.Anderson.Tools and !Users.Anderson.Utils. The For_Prefix is used to change the restored names.

Example 3

The following command restores all the archived error log files that were created in April 1987 and rebuilds them in the library !Users.Operator.Error_Logs. That is, the For_Prefix "!Machine" is replaced by the Use_Prefix "!Users.Operator." If !Users.Operator.Error_Logs doesn't exist, it will be created.

```
archive.restore (objects=>"!machine.error_logs.log_87_04_@",
                 use_prefix=>"!users.operator",
                 for_prefix=>"!machine");
```

Example 4

The following command restores the contents of the world !Users.Doyle.Tools into the world !Users.Anderson. However, only new objects are restored (objects that !Users.Anderson does not already contain).

```
archive.restore (objects=>"!users.doyle.tools.@",
                 use_prefix=>"!users.anderson",
                 for_prefix=>"!users.doyle.tools",
                 options=>"r1000,overwrite=new_objects");
```

See the introduction to this package for more examples.

References

procedure Save

---

procedure Save

```
procedure Save (Objects   : String := "<IMAGE>";
                Options   : String := "R1000";
                Device    : String := "MACHINE.DEVICES.TAPE_0";
                Response  : String := "<PROFILE>");
```

Description

Writes zero or more objects specified by the Objects and the Options parameters to the tape or library.

The place in which the objects are written is specified by the Device parameter.

The Save procedure:

- Preserves the library structure among multiple objects and records the unit state of Ada units.
- Saves objects recursively unless otherwise specified (see the Options parameter, below). In other words, the default is that, when the procedure is used to save a library, all the objects in that library are saved, including sublibraries and their contents. Similarly, when an Ada unit is archived, any subunits are also saved.
- Saves the links associated with each archived world and saves the name of the switch file associated with each archived library. However, the procedure does not automatically save the switch file.

The Save procedure writes all objects into a single file called Data. A file called Index is also created, which describes the name of each object in the Data file, the size of the object, and its unit state if it is an Ada unit. The Restore procedure uses the Index file to rebuild objects and place them in the appropriate states.

Parameters

```
Objects :  String := "<IMAGE>";
```

Specifies the object or objects to be saved. The parameter takes any naming expression. Objects can name objects such as worlds, directories, Ada units, text files, data files, switch files, activities, subsystems, and subsystem views. The default is the current image.

More than one object can be specified with the use of wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book).

```
Options : String := "R1000";
```

Specifies the options for the Save command. The Options parameter allows you to specify more specifically which objects to save and their access lists. For further information on specifying Options parameters, see "The Options Parameter" in the Key Concepts in this book).

The following list specifies the options available for this command:

`After=`***time***

Saves only objects that have been changed more recently than the specified ***time***, which can be a date, a time of day, or both, and can be written in any of the styles defined by the !Tools.Time_Utilities.Date_Format type and the !Tools.Time_Utilities.Time_Format type. For example, specifying the following value for the Options parameter saves only those objects named by Objects that were updated after June 11, 1987, at noon:

```
options=>"after=(06/11/87  12:00)"
```

`Compatibility_Database=`***subsystems***

Specifies that the full compatibility database for each specified subsystem should be archived. When Ada units in a subsystem are archived, the relevant portions of the subsystem compatibility database are automatically archived with them. Therefore, this option is required only in special situations, primarily when a primary and a secondary subsystem need to be synchronized. This option can be specified as "CDB", if desired. For further information on subsystems and compatibility databases, see Project Management (PM).

If no subsystems are specified in the ***subsystems*** portion of this option, the Options parameter specification is used instead. Any subsystem views specified by the Objects parameter will have the full compatibility database information saved for them.

The Nonrecursive option does not affect the interpretation of the CDB specification.

To archive compatibility databases only, use:

```
archive.save (objects=>"subsystems",
              options=>"cdb");
```

where ***subsystems*** is the name of the subsystems whose CDBs are to be saved.

To archive compatibility databases with other objects, use:

```
archive.save (objects=>"other",
              options=>"cdb=subsystems");
```

where ***other*** specifies objects that are probably disjoint from ***subsystems***.

`Format` literal

Specifies one of three formats for writing the Index and Data files. Format is irrelevant if Device is a library. If Device is a tape, the Format option must specify one of the following values:

R1000 Use the Format=R1000 option if the save operation requires only one tape. This format produces a single volume set formatted in chained ANSI:

— The volume set *DATA_SHORT* consists of a single tape that contains the Data file followed by the Index file. The images of the objects being saved are written directly onto tape.

R1000_Long Use the Format=R1000_Long option if the save operation requires more than one tape. This format produces two volume sets, both formatted in chained ANSI:

— The volume set *DATA_LONG* consists of multiple tapes that contain the Data file.

— The volume set *INDEX* consists of a single tape that contains the Index file.

Format=Ansi Use the Format=ANSI option if the save procedure is being performed to create a tape for a non-R1000 machine that accepts ANSI tapes. If you require this capability, see your Rational technical representative.

— This option writes the data into a temporary file and then writes both the Index and the Data files onto a tape using ANSI tape facilities.

— The volume set has no name and consists of a single tape that contains the Data file followed by the Index file.

Label=*string*

Specifies the string, used for identification purposes, that is written onto the tape at the head of the archived data, where *string* is any user-specified string. This label is not the same as the volume identifier or the volume set name. Typically, it used to record the tape owner's name, the date, and so on. The Restore procedure can be requested to verify the label on the tape. The verification is not case-sensitive. If the label doesn't match, the tape is rejected as if it were the wrong volume.

The label can be a single text line of arbitrary length. A *string* that contains special characters (commas or semicolons) must use balanced sets of parentheses to indicate that it is a string and thus should not be interpreted as special characters. For example, an option containing comma (,) characters could be specified as "Label=(MONDAY, JUNE 1, 1987)".

For further information on specifying strings in an Options parameter, see "The Options Parameter" in the Key Concepts in this book.

Nonrecursive Boolean

Specifies that only the objects that are actually named by the Objects parameter should be saved. This option does not recursively save sublibraries and subunits within the named objects. The option allows you to save the links and switch files associated with libraries without saving the objects in those libraries. It

also allows you to save a library plus a subset of its contents without saving the entire library.

This option prevents the subcomponents of libraries and Ada units from being implicitly saved. For example:

```
save (objects=>"[!users.hjl,!users.hjl.cli,!users.hjl.cli.@.]",
      options=>"r1ØØ nonrecursive");
```

saves only the named objects and not their substructures.

Prefix=***naming pattern***

Specifies a naming pattern that is saved with the archived objects. The pattern can be recalled as the For_Prefix parameter when the Restore procedure is used. When the option is set to an appropriate value, the restorer does not need to know the exact names of the archived objects to be able to restore them to a new place.

If the ***naming pattern*** value is not specified, the value is derived from the Objects parameter and the CDB option (if present). This is done by expanding context-sensitive characters (such as ~ and $), expanding indirect file references, and removing all attributes.

Version=***string***

The following options are of use to Rational personnel:

Version=GammaØ	Writes a tape that can be read on a Gamma 0 system.
Version=Gamma1	Writes a tape that can be read on a Gamma 1 system.
Version=***nnn***	Writes a tape that can be read by a previous version of package Archive (formerly called Source_Archive). The ***nnn*** represents a three-digit version number. For example: "version=21Ø".

```
Device : String := "MACHINE.DEVICES.TAPE_Ø";
```

Specifies the name of the device onto which the Index and Data files are to be written. The default is to write onto tape.

A library name can also be supplied, in which case the files are written into that library. If this library does not exist, it is created.

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

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

Example 1

The following library structure is referred to in this and the following examples:

```
!Example : Library;
  Source     : Library;
  Libraries  : Library;
  Logs       : Library;
  Tools      : Library;
  Move_List  : File;

!Example.Source : Library;
  File_One_Ada : File;

!Example.Libraries : Library;
  Utils   : Library;
  Misc    : Library;
  Example : Library;
  ...

!Example.Logs : Library;
  Parse_Log   : File;
  Make_Log    : File;

!Example.Tools : Library;
  Build     : Ada;
  Build     : Ada (Proc_Body);
  Parse_All : Ada;
  Parse_All : Ada (Proc_Body);
  ...
```

The following pair of commands saves the world !Example and all of its contents onto tape and then restores it in the library !Users.Anderson under the name New_Example. If !Users.Anderson does not exist, it is created.

From any context:

```
archive.save (objects=>"!example");
```

From any context:

```
archive.restore (objects=>"!example",
                 use_prefix=>"!users.anderson.new_example",
                 for_prefix=>"!example");
```

The Use_Prefix parameter is specified to tell the Archive.Restore command where to place the objects saved from !Example. It replaces the entire For_Prefix name. The new world name is !Users.Anderson.New_Example.

Example 2

Using the same library structure as in Example 1, the following pair of commands can be used to transfer the directories !Example.Libraries and !Example.Logs to another R1000 via tape. These directories are restored to the world !Users.Anderson. If !Users.Anderson does not already exist, it is created, and if the initiating user has operator capability, the user Anderson is created. The Objects parameter value is optional because there is only one object on the tape.

On the first R1000, from any context:

```
archive.save (objects=>"!example.1@");
```

On the second R1000, from any context:

```
archive.restore (objects=>"!example.1@",
                 use_prefix=>"!users.anderson",
                 for_prefix=>"!example");
```

The second R1000 now contains !Users.Anderson.Libraries and !Users.Anderson-.Logs.

Example 3

This example is the same as the previous one, except that it is assumed that !Users.Anderson exists and already contains directories called Libraries and Logs. The contents of !Example.Libraries and !Example.Logs will be merged with the contents of the existing !Users.Anderson.Libraries and !Users.Anderson.Logs. To prevent any objects from being overwritten, the New_Objects option is used in the Options parameter in the Restore procedure.

On the first R1000, from any context:

```
archive.save (objects=>"!example.1@");
```

On the second R1000, from any context:

```
archive.restore (objects=>"!example.1@",
                 use_prefix=>"!users.anderson",
                 for_prefix=>"!example"),
                 options=>"r1000,overwrite=new_objects");
```

Example 4

Again using the same library structure as in Example 1, the next pair of commands saves and restores recently updated objects contained in two directories. The directories are named using an indirect file, !Example.Move_List, which contains these entries:

```
!Example.Libraries.Utils
!Example.Libraries.Misc
```

Only objects that were modified after June 11, 1986, are to be saved, as specified by the Options parameter of the Save procedure. Furthermore, these objects are restored only if they will overwrite less recent versions, as specified by the Options parameter in the Restore procedure.

On the first machine, from any context:

```
archive.save (objects=>"_move_list",
              options=>"r1000,after=06/11/86);
```

On the second machine, from any context:

```
archive.restore (objects=>"?",
                 use_prefix=>"!users.anderson",
                 for_prefix=>"*",
                 options=>"r1000,overwrite=updated_objects,replace");
```

restores to !Users.Anderson.

See the introduction to this package for more examples.

---

end Archive;

---

package Compilation

With the operations in this package you can compile, demote, and destroy Ada units. You can also parse files containing Ada source code (uploaded from another machine) and create Ada units from them. The key procedures in this package are Parse, Promote, Make, Demote, and Destroy. This package also contains several types and procedures that provide underlying capabilities for these procedures.

The operations provided by this package can be performed on a number of Ada units or files at once through the use of wildcard characters or indirect files in the Unit_Name parameter.

Special Values

Some of the procedures in this package use *special values* in specifying a parameter of the Change_Limit subtype. Parameters of the Change_Limit subtype control which units an operation can modify, as follows:

"<SUBUNITS>"	Modifies the units named in the operation and their subunits.
"<UNITS>"	Modifies only the units named in the operation.
"<DIRECTORIES>"	Modifies only the units in the same set of directories as the units specified to the operation.
"<WORLDS>"	Modifies only the units in the same world as the units specified to the operation.
"<ALL_WORLDS>"	Modifies a unit in any world.

Compilation and Access Control

To promote a unit, you must have write access to that unit and write access to any unit it *with*s. For example, to promote unit Test, which *with*s units Driver and Data, you must have write access to all three units.

To demote a unit, you must have write access to that unit, but you need no access to units that *with* it. For example, to demote unit Test, which is *with*ed by Test_Driver, you must have write access to Test, but you require no access to Test_Driver.

Using Compilation with Rational Subsystems

The operations provided by this package can easily be used with subsystems since an activity file can be used to indirectly specify worlds to compile. Consider the following example.

A group of developers is working on an application called Program_Profile_System, which has been decomposed into three subsystems. This application analyzes Ada code, collects statistics, and generates reports. The topmost subsystem is called Report_Layer, the middle subsystem is called System_Layer, and the lowest level is called Unit_Layer. Both the System_Layer and the Report_Layer import the spec view of the Unit_Layer subsystem.

If a spec-incompatible change (for example, changing the name of a function) is made to the Unit_Layer subsystem, the other two subsystems in this application will require changes to all using occurrences of this function. Once all of these changes are made, it will be useful to recompile everything in the system using the Make procedure. Using an activity file as an indirect file for the Unit_Name parameter can achieve this goal. Activity files can also be used similarly to wildcard characters or attributes to operate on a large number of objects at one time.

For information on the use of subsystems, see Project Management (PM).

Special Names

Many of the commands in this package have *special names* as default values to parameters requiring names. Anywhere that a string name can be used, a special name can be used. Special names allow you to designate without supplying a pathname. They take the form "*<special name>*", where *special name* specifies text, an object, or a region, as described below:

"<SELECTION>"	References the object associated with the highlighted area, when the cursor is located in a highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the object on which the cursor is located, whether or not there is a highlighted area in the window.
"<IMAGE>"	References the highlighted object, if the cursor is in a highlighted area. If the cursor is not located in the highlighted area, this special name references the image in which the cursor is located.
"<TEXT>"	References the object named in the highlighted text in the image in the window.
"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, this special name references that activity rather than the default activity.

You can replace special names with other types of naming expressions, as accepted by that parameter.

Error Response

The commands in this package have a Response parameter that specifies how the command should respond to errors, how to generate logs, and what activities to use. The response profile "<PROFILE>", which many commands use by default, specifies the job response profile. If there is no job response profile, the session response profile ("<SESSION_PROFILE>") is used. If there is no session response profile, the system's default profile ("<DEFAULT>") is used. For further information on profiles, see SJM, package Profile.

constant All_Worlds

```
All_Worlds : constant Change_Limit := "<ALL_WORLDS>";
```

Description

Defines a value of the Change_Limit type that indicates that units in any world may change when the operations in this package are performed.

This constant is retained for compatibility with previous releases. All_Worlds is equivalent to the special value "<ALL_WORLDS>".

References

subtype Change_Limit

constant Current_Directory

constant Same_Directories

constant Same_World

constant Same_Worlds

procedure Atomic_Destroy

```
procedure Atomic_Destroy (Unit       :     Unit_Name;
                          Success    : out Boolean;
                          Action_Id  :     Action.Id     := Action.Null_Id;
                          Limit      :     Change_Limit  := "<WORLDS>";
                          Response   :     String        := "<PROFILE>");
```

Description

Destroys the named object and any dependent units.

This procedure is a special case of the Destroy procedure. This procedure should not be used without consultation with your Rational technical representative. Use the Destroy procedure instead.

subtype Change_Limit

```
subtype Change_Limit is String;
```

Description

Defines which units are allowed to change when an operation in this package is performed.

Many of the operations in this package require a parameter of the Change_Limit type. Parameters of the Change_Limit type control which units an operation can modify. This parameter has five predefined special values with the following meanings:

"<SUBUNITS>"	Modifies the units named in the operation and their subunits.
"<UNITS>"	Modifies only the units named in the operation.
"<DIRECTORIES>"	Modifies only the units in the same set of directories as the units specified to the operation.
"<WORLDS>"	Modifies only the units in the same world as the units specified to the operation.
"<ALL_WORLDS>"	Modifies a unit in any world.

Each change limit includes the units implied by the Change_Limit type above it in this list.

Any unique prefix of the special value enclosed in quotation marks and brackets ("<>") is recognized. Thus "<W>", "<WORLD>", and "<WORLDS>" are all valid, equivalent ways of specifying "<WORLDS>".

A Change_Limit parameter may also be a string name that designates a set of worlds or directories. When names using strings are specified, only the units in the specified worlds or directories are allowed to change. Context prefixes, wildcards, indirect files, special names, and attributes can be used in specifying object names.

An activity file can also be used in a string name as an indirect file. This is useful when working with subsystems.

References

constant All_Worlds

constant Current_Directory

constant Same_Directories

constant Same_World

constant Same_Worlds

---

procedure Compile

```
procedure Compile (File_Name       : Name          := "<REGION>";
                   Library         : Name          := "$";
                   Goal            : Unit_State    := Compilation.Installed;
                   List            : Boolean       := False;
                   Source_Options  : String        := "";
                   Limit           : Change_Limit  := "<WORLDS>";
                   Response        : String        := "<PROFILE>");
```

Description

Compiles the specified text file into the specified library.

This procedure parses and promotes the units in the specified file or files to the specified goal state. A listing with interleaved error messages is produced in the log file if the List parameter is true. If there are any errors, the unit is not added to the library.

This procedure implements LRM semantics for compilation into Ada libraries and is used primarily for running the Ada language validation tests.

Parameters

`File_Name : Name := "<REGION>";`

Specifies the file that contains the units to be compiled. Special names, indirect files, wildcards, context prefixes, and attributes are allowed in this name. The default special name `"<REGION>"` specifies the highlighted object.

`Library : Name := "$";`

Specifies the library into which the units will be compiled. The default is the current library.

`Goal : Unit_State := Compilation.Installed;`

Specifies the desired state for the compiled units. The default is the installed state.

`List : Boolean := False;`

Specifies whether to produce a listing in the log file. The default is not to produce a listing.

```
Source_Options : String := "";
```

Specifies a series of options for the compilation. This parameter currently is not implemented and is reserved for future development.

```
Limit : Change_Limit := "<WORLDS>";
```

Specifies the limit to the scope of changes. The default is to change units only in the worlds of those units specified.

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

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

Errors

If the names of the source text files are identical to the name of the Ada unit they contain, an error will occur. Two objects with the same name cannot exist in the same library. A suggested strategy is to keep text files in one directory and compile Ada units into another library.

Example

The command:

```
compilation.compile ("text", "my_library", compilation.coded, true);
```

creates an Ada unit in the library, My_Library, using the text file called Text as input.

References

subtype Change_Limit

subtype Name

subtype Unit_State

---

constant Current_Directory

```
Current_Directory : constant Change_Limit := Same_Directories;
```

Description

Defines a constant that can be used as a value for a parameter of the Change_Limit type.

The Change_Limit parameter controls which units an operation is allowed to modify.

Use of the Current_Directory constant as a parameter specifies that only the units in the same directory as the units specified to the operation are allowed to change.

This constant is retained for compatibility with previous releases. Current_Directory is equivalent to the special value "<DIRECTORIES>".

References

constant All_Worlds

subtype Change_Limit

constant Same_Directories

constant Same_World

constant Same_Worlds

procedure Delete

```
procedure Delete (Unit     : Unit_Name    := "<SELECTION>";
                  Limit    : Change_Limit := "<WORLDS>";
                  Response : String       := "<PROFILE>");
```

Description

Demotes and deletes the default version of the specified object and any subunits.

This procedure deletes the named object by first demoting any dependent units. This is a useful capability when deleting Ada units. The deletions are reversible in the sense that specified objects can be undeleted with the Library.Undelete procedure. This command is different from the Destroy procedure, which permanently deletes and expunges objects.

Subordinate units (bodies and subunits) are also deleted. If the deletion of the object or any subordinate object would obsolesce units outside the specified change limit, the operation will fail.

Parameters

`Unit : Unit_Name := "<SELECTION>";`

Specifies the name of the object to be deleted. Special names, wildcards, context prefixes, indirect files, and attributes are allowed in this name. The default is the current selection.

`Limit : Change_Limit := "<WORLDS>";`

Specifies which units can be demoted to allow the deletion. The default is to allow demotion of units only in the worlds that contain a unit to be deleted.

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

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

Errors

Common errors include specification of a Limit parameter that does not include all units that must be demoted.

A Lock_Error can be caused by editing a unit that needs to be demoted.

Access errors can be caused by the job not having the proper class of access for an object.

Example

Consider the following world:

```
!Users.Gzc.Test_World.Lifecycle_Example  : Library (World);
  Complex                 : Ada (Pack_Spec);
  Complex                 : Ada (Pack_Body);
  Complex_Utilities       : Ada (Pack_Spec);
  Complex_Utilities       : Ada (Pack_Body);
  Display_Complex_Sums    : Ada (Proc_Spec);
  Display_Complex_Sums    : Ada (Proc_Body);
  List_Generic            : Ada (Gen_Pack);
  List_Generic            : Ada (Pack_Body);
  Sample_Input            : File;
  Srs                     : File;
```

The command:

```
compilation.delete
         (unit=>"complex'spec", limit=>"<WORLDS>",
          response=>"<PROFILE>");
```

deletes both the spec and the body of package Complex and demotes any dependent units. The body is deleted because it is a subordinate object that relies on the existence of the spec. If a Library.Delete procedure were attempted to delete the spec, it would fail because of this dependency.

References

subtype Change_Limit

procedure Destroy

subtype Unit_Name

---

procedure Demote

```
procedure Demote (Unit         : Unit_Name    := "<SELECTION>";
                  Goal         : Unit_State   := Compilation.Source;
                  Limit        : Change_Limit := "<WORLDS>";
                  Effort_Only  : Boolean      := False;
                  Response     : String       := "<PROFILE>");
```

Description

Demotes the specified units to the specified goal state, demoting any other units, within the limit necessary to achieve the requested demotion.

The procedure demotes the specified unit and all of its dependents in such a way that the entire system is always semantically consistent. The procedure finds the dependents of the specified unit and all of the dependents of those units until the entire transitive closure of the specified unit is known. Then the procedure demotes the dependent units followed by the specified unit to the goal state.

The demotion request fails if any units outside the specified limit must be demoted. Conversely, all units demoted by this procedure are within that limit.

If the current state of the unit or any of its dependents is the same or lower than the goal state, the procedure has no effect on that unit or dependent units.

The procedure can also estimate the amount of work necessary to accomplish a specified demotion. The Effort_Only parameter can specify that the procedure only estimate the amount of work in doing a specified demotion without actually demoting any units.

This procedure is useful when editing an object that has dependents. The !Commands.Common.Demote procedure does not demote the dependents and fails if dependents exist. In that case, this procedure can be used to demote the unit and all of the dependents so that the unit can be edited. Then the program can be promoted again with the Promote procedure or the Make procedure.

Parameters

```
Unit :  Unit_Name := "<SELECTION>";
```

Specifies the name of the Ada unit to be demoted. Special names, wildcards, context prefixes, indirect files, and attributes are allowed in this name. The default is the current selection.

```
Goal : Unit_State := Compilation.Source;
```

Specifies the desired state of the unit. The default demotes the unit to the source state.

```
Limit : Change_Limit := "<WORLDS>";
```

Specifies which units can be demoted. The default is to allow demotion of units only in the worlds that contain the specified unit to be demoted.

```
Effort_Only : Boolean := False;
```

Specifies whether to check for the effort required to do the demotion and the subsequent promotion. The effort rating reported is a relative measure of the amount of work involved. The default is to check and to actually do the demotion.

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

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

Errors

Common errors include specification of a Limit parameter that does not include all of the units that must be demoted.

A Lock_Error can be caused by editing a unit that needs to be demoted.

Access errors can be caused by the job not having the proper class of access for an object.

Example

Consider the following world:

```
!Users.Gzc.Lifecycle_Example  : Vol 4  1;
  Complex                 : C 86/12/29 15:16:37 Gzc      9643  1    ;
  Complex                 : C 86/12/29 15:16:55 Gzc     10383  1    ;
  Complex_Utilities       : C 86/12/29 15:34:00 Gzc     15910  1    ;
  Complex_Utilities       : C 86/12/29 15:33:47 Gzc     26192  1    ;
  Display_Complex_Sums    : C 86/12/29 15:34:06 Gzc      7166  1    ;
  Display_Complex_Sums    : C 86/12/29 15:33:51 Gzc     23587  1    ;
  List_Generic            : C 86/12/29 15:34:20 Gzc     15423  1    ;
  List_Generic            : C 86/12/29 15:34:16 Gzc     39080  1    ;
```

The command:

```
compilation.demote
         (unit=>"complex'spec", goal=>compilation.source,
          limit=>"<WORLDS>", effort_only=>false,
          response=>"<PROFILE>");
```

demotes the unit and displays the following in the current output window:

```
------------------------------------------------------------------------

!USERS.GZC.LIFECYCLE_EXAMPLE  % COMPILATION.DEMOTE       STARTED 12:37:29 PM
------------------------------------------------------------------------


87/03/20 12:37:33 ::: [Compilation.Demote ("complex'spec", SOURCE, "<WORLDS>",
87/03/20 12:37:33 ...  FALSE, PERSEVERE);].
87/03/20 12:37:34 --- Attempting to demote !USERS.GZC.LIFECYCLE_EXAMPLE.
87/03/20 12:37:34 ... COMPLEX.
87/03/20 12:37:34 +++ !USERS.GZC.LIFECYCLE_EXAMPLE.COMPLEX'BODY'V(1)  demoted
87/03/20 12:37:35 ... to SOURCE.
87/03/20 12:37:35 +++ !USERS.GZC.LIFECYCLE_EXAMPLE.
87/03/20 12:37:35 ... COMPLEX_UTILITIES'BODY'V(1)  demoted to SOURCE.
87/03/20 12:37:36 +++ !USERS.GZC.LIFECYCLE_EXAMPLE.
87/03/20 12:37:36 ... DISPLAY_COMPLEX_SUMS'BODY'V(1)  demoted to SOURCE.
87/03/20 12:37:36 +++ !USERS.GZC.LIFECYCLE_EXAMPLE.COMPLEX_UTILITIES'V(1)
87/03/20 12:37:36 ... demoted to SOURCE.
87/03/20 12:37:37 +++ !USERS.GZC.LIFECYCLE_EXAMPLE.COMPLEX'V(1)  demoted to
87/03/20 12:37:37 ... SOURCE.
87/03/20 12:37:38 ::: [End of Compilation.Demote Command].
```

References

subtype Change_Limit

constant Source

subtype Unit_Name

type Unit_State

---

procedure Dependents

```
procedure Dependents (Unit       : Unit_Name := "<IMAGE>";
                      Transitive : Boolean   := False;
                      Response   : String    := "<PROFILE>");
```

Description

Displays the set of units that depend on the current or named units.

This procedure produces a log that lists the set of units that depend on this unit. That list can be used to approximate the significance of changing this unit. The list can include the direct dependents or the entire set of dependents (the transitive closure of dependents).

Parameters

`Unit : Unit_Name := "<IMAGE>";`

Specifies the units whose dependents are desired. Special names, wildcards, context prefixes, indirect files, and attributes are allowed in this name. The default is the currently selected object or, if no object is selected, the current image.

`Transitive : Boolean := False;`

Specifies whether to list the transitive closure of all dependents or just the direct dependents. The default is the direct dependents.

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

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

Restrictions

The specified unit and its dependents must be at the installed or coded state for this procedure to produce the expected results. Dependencies do not exist on units in the source state and will not be listed in the output.

Errors

Errors include specification of the incorrect unit name.

Example

Consider the following world:

```
!Users.Gzc.Test_World.Lifecycle_Example  : Vol 4  1;
  Complex                : C 86/12/23 17:33:47 Gzc      9643  1    ;
  Complex                : C 86/12/23 17:34:10 Gzc     10383  1    ;
  Complex_Utilities      : C 86/12/23 17:33:54 Gzc     15910  1    ;
  Complex_Utilities      : C 86/12/23 17:31:58 Gzc     26192  1    ;
  Display_Complex_Sums   : C 86/12/23 17:32:26 Gzc      7166  1    ;
  Display_Complex_Sums   : C 86/12/23 17:31:54 Gzc     23587  1    ;
  List_Generic           : C 86/12/23 17:34:00 Gzc     15423  1    ;
  List_Generic           : C 86/12/23 17:32:17 Gzc     39080  1    ;
  Sample_Input           :   86/11/07 17:18:54 *System    39  1    ;
  Srs                    :   86/11/07 17:18:55 *System  4552  1    ;
```

The command:

```
compilation.dependents
         (unit=>"complex", transitive=>false,
          response=>"<PROFILE>");
```

produces the following display in the current output window, indicating that the spec of package Complex has four dependents and the body has none:

```
------------------------------------------------------------------------

GZC.TEST_WORLD.LIFECYCLE_EXAMPLE  % COMPILATION.DEPENDENTS  STARTED 5:36:29 PM
------------------------------------------------------------------------


86/12/23 17:36:30 ::: [Compilation.Dependents ("complex", FALSE, PERSEVERE);].
!USERS.GZC.TEST_WORLD.LIFECYCLE_EXAMPLE.COMPLEX'V(13)  has the following
dependents:
  !USERS.GZC.TEST_WORLD.LIFECYCLE_EXAMPLE.COMPLEX'BODY'V(12)
  !USERS.GZC.TEST_WORLD.LIFECYCLE_EXAMPLE.COMPLEX_UTILITIES'V(6)
  !USERS.GZC.TEST_WORLD.LIFECYCLE_EXAMPLE.COMPLEX_UTILITIES'BODY'V(4)
  !USERS.GZC.TEST_WORLD.LIFECYCLE_EXAMPLE.DISPLAY_COMPLEX_SUMS'BODY'V(4)
86/12/23 17:36:32 --- !USERS.GZC.TEST_WORLD.LIFECYCLE_EXAMPLE.
86/12/23 17:36:32 ... COMPLEX'BODY'V(12) has no dependents.
86/12/23 17:36:32 ::: [End of Compilation.Dependents Command].
```

References

subtype Unit_Name

procedure Destroy

```
procedure Destroy (Unit       : Unit_Name    := "<SELECTION>";
                   Threshold  : Natural      := 1;
                   Limit      : Change_Limit := "<WORLDS>";
                   Response   : String       := "<PROFILE>");
```

Description

Destroys the named object and any subordinate units and demotes dependent units.

This procedure deletes and then expunges the named unit. Unlike the Delete procedure, this procedure is not reversible. Once a unit is destroyed, it cannot be recovered.

If the deletion of the unit would orphan any subordinate units, those units are also destroyed. If the deletion of the unit or any subordinate unit would obsolesce units outside the named limit, the operation fails.

The procedure provides a threshold number of dependent units to destroy. If the number of the units to destroy is greater than the threshold, the procedure abandons all destructions. The procedure can be executed again with a new threshold.

Parameters

`Unit : Unit_Name := "<SELECTION>";`

Specifies the name of the unit to be destroyed. Special names, wildcards, context prefixes, indirect files, and attributes are allowed in this name. The default is the current selection.

`Threshold : Natural := 1;`

Specifies the number of units per specified unit that can be destroyed before the procedure fails. The default permits just the named unit, but no dependents or subordinates, to be destroyed.

`Limit : Change_Limit := "<WORLDS>";`

Specifies which units can be demoted as a side effect of the destroy operation. The default allows units in the same world as the units being destroyed to be demoted.

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

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

Errors

Common errors include specification of a Limit parameter that does not include all units that must be demoted and incorrect specification of the Threshold parameter.

A Lock_Error can be caused by editing a unit that needs to be destroyed.

Access errors can be caused by the job not having the proper class of access for a unit.

Example

Consider the following world:

```
!Users.Gzc.Test_World.Lifecycle_Example  : Library (World);
  Complex                : Ada (Pack_Spec);
  Complex                : Ada (Pack_Body);
  Complex_Utilities      : Ada (Pack_Spec);
  Complex_Utilities      : Ada (Pack_Body);
  Display_Complex_Sums   : Ada (Proc_Spec);
  Display_Complex_Sums   : Ada (Proc_Body);
  List_Generic           : Ada (Gen_Pack);
  List_Generic           : Ada (Pack_Body);
  Sample_Input           : File;
  Srs                    : File;
```

The command:

```
compilation.destroy
         (unit=>"complex'spec",  threshold=>1,
          limit=>"<WORLDS>",
          response=>"<PROFILE>");
```

destroys both the spec and the body of package Complex and demotes their dependents. This destruction is irreversible. It also demotes any dependent objects.

References

subtype Change_Limit

subtype Unit_Name

---

renamed procedure Make

```
procedure Make (Unit         : Unit_Name      := "<!MAGE>";
                Scope        : Promote_Scope  := Compilation.All_Parts;
                Goal         : Unit_State     := Compilation.Coded;
                Limit        : Change_Limit   := "<WORLDS>";
                Effort_Only  : Boolean        := False;
                Response     : String         := "<PROF!LE>") renames Promote;
```

Description

Promotes the specified units to the specified goal state.

By default, this procedure promotes the units, their subordinates, and the specs, bodies, and subunits of all units they depend on to the coded state. This procedure is typically used when you want to complete the compilation necessary to get the specified interface ready to execute. If you are just checking a unit for semantic consistency and do not need to execute, use the Promote procedure to minimize time and effort.

The procedure is applied recursively to the named unit and any other units in the compilation closure of the unit as specified by the Scope parameter. The procedure promotes the units to the goal state if possible. (A unit is not promoted if it is not a legal Ada unit.) If any unit is already at or above the goal state, the procedure has no effect on that unit. The correct order of compilation is determined by the Make procedure.

The promotion request fails if any units outside the specified limit must be promoted. Conversely, all units promoted by this procedure are within the specified limit.

The procedure can also estimate the amount of work necessary to accomplish a specified promotion. The Effort_Only parameter can specify that the procedure only estimate the amount of work in doing a specified promotion without actually promoting any units.

Parameters

```
Unit :  Unit_Name := "<!MAGE>";
```

Specifies the name of the unit to be promoted. Special names, wildcards, context prefixes, indirect files, activities, and attributes are allowed in this name. The default is the current image.

```
Scope : Promote_Scope := Compilation.All_Parts;
```

Specifies the scope of the promotion. The default is to promote the visible part and the body of the named units and their subunits and recursively all units in *with* clauses and their transitive closure.

```
Goal : Unit_State := Compilation.Coded;
```

Specifies the desired state of the units after the promotion. The default is to promote to the coded state.

```
Limit : Change_Limit := "<WORLDS>";
```

Specifies the units that may be promoted. The default allows units in the same worlds as the specified units to be promoted.

```
Effort_Only : Boolean := False;
```

Specifies whether to check for the effort required to do the promotion. The effort rating reported is a relative measure of the amount of work involved. The default is to check and to actually do the promotion.

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

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

Errors

Common errors include specification of a Limit parameter that does not include all units that must be promoted.

A Lock_Error can be caused by editing a unit that needs to be promoted.

Access errors can be caused by the job not having the proper class of access for an object.

Example

Consider the following world and the state of the objects within that world:

```
!Users.Gzc.Test_World.Lifecycle_Example  : Vol 4  1;
  Complex                : C 86/12/23 15:41:23 Gzc      9643  1    ;
  Complex                : C 86/12/23 15:41:19 Gzc     10383  1    ;
  Complex_Utilities      : S 86/12/23 15:41:22 Gzc     15910  1    ;
  Complex_Utilities      : S 86/12/23 15:41:20 Gzc     26192  1    ;
  Display_Complex_Sums   : S 86/12/23 15:41:23 Gzc      7166  1    ;
  Display_Complex_Sums   : S 86/12/23 15:41:21 Gzc     23587  1    ;
  List_Generic           : S 86/12/23 15:41:25 Gzc     15423  1    ;
  List_Generic           : S 86/12/23 15:41:25 Gzc     39080  1    ;
```

The command:

```
compilation.make
     (unit=>"!users.gzc.test_world.lifecycle_example",
      scope=>compilation.all_parts,
      goal=>compilation.coded,  limit=>"<WORLDS>",
      effort_only=>false,  response=>"<PROFILE>");
```

causes all of the units in that library to be promoted to the coded state.

References

subtype Change_Limit

procedure Promote

subtype Unit_Name

type Unit_State

subtype Name

```
subtype Name is String;
```

Description

Defines the form of names used as parameters in procedures in this package.

These names can use the special names, wildcards, context prefixes, indirect files, and attributes that are discussed in the Key Concepts in this book.

procedure Parse

```
procedure Parse (File_Name       : Name    := "<REGION>";
                 Directory       : Name    := "$";
                 List            : Boolean := False;
                 Source_Options  : String  := "";
                 Response        : String  := "<PROFILE>");
```

Description

Parses the Ada source in the specified files and creates corresponding Ada units in the specified directory.

This procedure is most useful when transporting Ada source code from another host to the Rational Environment. In that case, source code is loaded into text files in a directory in the Environment. These files are then transformed into parsed Ada units using this procedure.

The procedure scans the contents of the named file, searching for compilation units. The procedure determines the correct parsing order of those units, builds the unit declarations in the context, and then parses the subunits.

Parameters

`File_Name : Name := "<REGION>";`

Specifies the name of the file that contains the unit or units to be parsed. Special names, wildcards, context prefixes, indirect files, and attributes are allowed in this name. The default is the selected region.

`Directory : Name := "$";`

Specifies the name of the directory or world in which to create the parsed units. Special names, wildcards, context prefixes, indirect files, and attributes are allowed in this name. This name must resolve to one library object; it must be unique. The default is the current directory.

`List : Boolean := False;`

Specifies whether to list each unit, as parsed, in the log file. The default is not to list each unit.

```
Source_Options :  String := "";
```

Specifies a series of options for the compilation. This is reserved for future enhancements.

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

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

Errors

If the names of the source text files are identical to the name of the Ada unit they contain, an error will occur. Two objects with the same name cannot exist in the same library. A suggested strategy is to keep text files in one directory and compile Ada units into another library.

Example

Consider the following world:

```
!Users.Gzc.Test_World.Lifecycle_Example  : Library (World);
  Complex_Utilities      : Ada (Pack_Spec);
  Complex_Utilities      : Ada (Pack_Body);
  Display_Complex_Sums   : Ada (Proc_Spec);
  Display_Complex_Sums   : Ada (Proc_Body);
  List_Generic           : Ada (Gen_Pack);
  List_Generic           : Ada (Pack_Body);
  Sample_Input           : File;
  Srs                    : File;
  File_1                 : File (Text);
```

The command:

```
compilation.parse ("file_1");
```

takes a text file called File_1 containing the package spec for an Ada unit called Complex and creates a package spec called Complex.

References

subtype Name

---

procedure Promote

```
procedure Promote (Unit          : Unit_Name      := "<IMAGE>";
                   Scope         : Promote_Scope  := Compilation.Subunits_Too;
                   Goal          : Unit_State     := Compilation.Installed;
                   Limit         : Change_Limit   := "<WORLDS>";
                   Effort_Only   : Boolean        := False;
                   Response      : String         := "<PROFILE>");
```

Description

Promotes the specified unit in the specified scope to the specified goal state.

This procedure is applied recursively to the named unit and to any other units in the specified scope. A unit is not promoted if it is not a legal Ada unit. The procedure promotes the units to the goal state if possible. If any unit is already at or above the goal state, the procedure has no effect on that unit. If you are just checking a unit for semantic consistency and do not need to execute, use this procedure. If you want to execute, use the Make renamed procedure.

The promotion request fails if any units outside the specified limit must be promoted. Conversely, all units promoted by this procedure are within the specified limit.

The procedure can also estimate the amount of work necessary to accomplish a specified promotion. The Effort_Only parameter can specify that the procedure only estimate the amount of work in doing a specified promotion without actually promoting any units.

The procedure promotes units individually. Each promotion of a unit is a separate action. If the unit is to be promoted by more than one state (from source to coded), the promotion from one state to the next is done as a separate action (source to installed, then installed to coded). If the promotion of a particular subunit fails because of semantic errors or other reasons, the promotion of any other units may not be affected, depending on the error reaction specified.

The procedure does not lock any units before promoting them. Thus, if one job is promoting some units and another job is demoting some of the same units, one or the other job will have errors. If two jobs are promoting the same units, the first job may fail while the second job succeeds.

By default, this procedure will promote the named units and any of their subunits and units they *with* to the installed state. If the named unit is a visible part, the corresponding body and the bodies of any subunits are not promoted. If the named unit is a body, the corresponding visible part is promoted, and the bodies of any of its subunits are promoted. If the named unit has other units in its *with* clause, the visible parts of those other units are promoted also.

The Make renamed procedure renames this procedure but uses different default parameter values.

Parameters

```
Unit : Unit_Name := "<IMAGE>";
```

Specifies the name of the unit to be promoted. Special names, wildcards, indirect files, context prefixes, indirect files, and attributes are allowed in this name. The default is the current image if there is no selection. Indirect filenames must be prefaced by an underscore character.

```
Scope : Promote_Scope := Compilation.Subunits_Too;
```

Specifies the scope of the promotion. See "Description," above, for further information.

```
Goal : Unit_State := Compilation.Installed;
```

Specifies the desired state of the units after the promotion. The default is to promote to the installed state.

```
Limit : Change_Limit := "<WORLDS>";
```

Specifies the units that can be promoted. The default allows units in the same worlds as the specified units to be promoted.

```
Effort_Only : Boolean := False;
```

Specifies whether to check for the effort required to do the promotion. The effort rating reported is a relative measure of the amount of work involved. The default is to check and to actually do the promotion.

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

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

Errors

Common errors include specification of a Limit parameter that does not include all units that must be promoted.

A Lock_Error can be caused by editing a unit that needs to be promoted.

Access errors can be caused by the job not having the proper class of access for an object.

Example

Consider the following world and the state of the Ada units within that world:

```
!Users.Gzc.Lifecycle_Example  : Vol 4  1;
  Complex                 : S 86/12/23 17:40:25 Gzc       9643  1     ;
  Complex                 : S 86/12/23 17:40:21 Gzc      10383  1     ;
  Complex_Utilities       : S 86/12/23 17:40:24 Gzc      15910  1     ;
  Complex_Utilities       : S 86/12/23 17:40:22 Gzc      26192  1     ;
  Display_Complex_Sums    : S 86/12/23 17:40:25 Gzc       7166  1     ;
  Display_Complex_Sums    : S 86/12/23 17:40:23 Gzc      23587  1     ;
  List_Generic            : S 86/12/23 17:40:27 Gzc      15423  1     ;
  List_Generic            : S 86/12/23 17:40:27 Gzc      39080  1     ;
  Sample_Input            :   86/11/07 17:18:54 *System     39  1     ;
  Srs                     :   86/11/07 17:18:55 *System   4552  1     ;
```

The command:

```
compilation.promote
         (unit=>"!users.gzc.lifecycle_example",
          scope=>compilation.subunits_too,
          goal=>compilation.installed,  limit=>"<worlds>",
          effort_only=>false,  response=>"<PROFILE>");
```

causes all of the units in that world to be promoted to the installed state.

References

subtype Change_Limit

procedure Make

subtype Name

type Unit_State

type Promote_Scope

```
type Promote_Scope is (Single_Unit, Unit_Only, Subunits_Too,
                       All_Parts, Load_Views);
```

Description

Defines the scope of units to be promoted, besides the named units, in a promote operation.

The units in the Promote_Scope type will be promoted even if they do not need to be promoted to promote the named units. The exception to this is the Single_Unit enumeration, which promotes only the named unit.

Enumerations

`All_Parts`

Specifies that all parts of the named unit and any subunits be promoted. If the named unit is a visible part, both the visible part and the body are promoted. If the named unit is a body, it and the corresponding visible part are promoted. In all cases, all subunits are promoted as well. If the unit has other units in its ***with*** clause, the visible parts, bodies, and subunits of those other units are promoted also.

`Load_Views`

Specifies that load views associated with each named spec view be promoted. In all cases, all load view units referenced by the activity for the spec view are promoted. If the load view units have other units in their ***with*** clauses, the visible parts, bodies, and subunits of those other units are promoted also.

`Single_Unit`

Specifies that the named unit only be promoted. Any subunits in this unit are not promoted. If the unit has other units in its ***with*** clause, those units are not promoted. If the named unit is a visible part, the body is not promoted. If the named unit is a body, the corresponding visible part is not promoted. Use this if you know that all of the ***with***ed units are installed.

Subunits_Too

Specifies that the named unit and any of its subunits and units they *with* be promoted. If the named unit is a visible part, the corresponding body and the bodies of any subunits are not promoted. If the named unit is a body, the corresponding visible part is promoted, and the bodies of any of its subunits are promoted. If the named unit has other units in its *with* clause, the visible parts of those other units are promoted also.

Unit_Only

Specifies that the named unit and any units it *with*s be promoted. Any subunits in this unit are not promoted. If the unit has other units in its *with* clause, the visible part of those units are promoted. Both visible part and body, if applicable, are promoted.

References

procedure Make

procedure Promote

---

constant Same_Directories

```
Same_Directories : constant Change_Limit := "<DIRECTORIES>";
```

Description

Defines a value of the Change_Limit type that indicates that units in the same set of directories may change when operations in this package are performed.

This constant is retained for compatibility with previous releases. The Same_Directories constant is equivalent to the special value "<DIRECTORIES>".

References

constant All_Worlds

subtype Change_Limit

constant Current_Directory

constant Same_World

constant Same_Worlds

constant Same_World

```
Same_World : constant Change_Limit := Same_Worlds;
```

Description

Defines a value of the Change_Limit type that indicates that units in any world may change when the operations in this package are performed.

This constant is retained for compatibility with previous releases. The Same_World constant is equivalent to the special value "<WORLDS>".

References

constant All_Worlds

subtype Change_Limit

constant Current_Directory

constant Same_Directories

constant Same_Worlds

constant Same_Worlds

```
Same_Worlds : constant Change_Limit := "<WORLDS>";
```

Description

Defines a value of the Change_Limit type that indicates that units in any world may change when the operations in this package are performed.

This constant is retained for compatibility with previous releases. The Same_Worlds constant is equivalent to the special value "<WORLDS>".

References

constant All_Worlds

subtype Change_Limit

constant Current_Directory

constant Same_Directories

constant Same_World

subtype Unit_Name

```
subtype Unit_Name is String;
```

Description

Defines the name for a unit or set of units to be supplied to a compilation command.

A parameter of the Unit_Name type may designate a set of Ada units, worlds, directories, or activities. If a world or directory is specified, all Ada units contained by that world or directory are operated on. Using an activity file as an indirect file allows easy specification of the views specified by that activity.

A unit can be put in archived state to save space. Space used by an archived unit is about 10% of that required by a coded unit.

type Unit_State

```
type Unit_State is (Archived, Source, Installed, Coded);
```

Description

Defines the compilation states in which an Ada unit can be.

This type defines four states. These states are ordered—that is, a unit must progress from source through installed to coded before it can execute. The nondefault versions of an Ada unit are saved in archived state to save space. These files must be promoted to source before they can be edited.

A unit can be put in archived state to save space. The space used by an archived unit is about 10% of that required by a coded unit.

Enumerations

Archived

Defines a unit state. Units in the archived state are much more compact in size than units in other states. A unit in this state cannot be edited. It must be promoted from archived to source state first. Units in this state also do not have the Definition capability and object-oriented highlighting available to units in the source, installed, and coded states.

Coded

Defines a unit state. A unit in this state is completely compiled; it is syntactically and semantically correct and has machine code generated for it. The unit is known to the Environment and may have semantic dependencies.

Installed

Defines a unit state. A unit in this state is syntactically and semantically correct. Such a unit can be referenced by any other unit according to the Ada visibility rules. The unit is known to the Environment and is controlled by the Environment to prevent deletions or changes that would alter the semantic content of the unit or units that reference it.

Source

Defines a unit state. This is the state in which all units are created. A unit in this state may or may not be semantically or syntactically correct. Procedures exist in the Environment to help make a unit syntactically and semantically correct. The !Commands.Common.Format procedure provides syntactic analysis and completion. The !Commands.Common.Semanticize procedure provides semantic analysis.

A unit in source state is not semantically known to other units. This means that none of the types, procedures, and functions that are exported from this unit can be used or called by any other installed unit. No other unit can check its semantic validity against this interface.

---

end Compilation;

---

package File_Utilities

Package File_Utilities provides a set of subprograms that allow any object that can be opened for text I/O (for example, Ada units and files) to be compared, merged, and searched.

This package also provides pattern matching, which is useful in comparing a file against a template file. For example, a user has written a program that should produce output of the format: the letter A or B, followed by a two-digit integer. The user could create a pattern file containing the following pattern:

```
[AB][Ø123456789][Ø123456789]
```

The user could then use the Compare procedure or the Equal function to compare the output file to the pattern file.

Patterns can be specified in the Pattern parameter of the Find procedure and the Found function as well.

The Compare procedure and the Equal function have an Options parameter that provides additional flexibility in performing file comparisons. This capability is useful in automating the comparison of software test results against desired results and in software documentation.

Special Names

Many of the commands in this package have *special names* as default values to parameters requiring names. Anywhere that a string name can be used, a special name can be used. Special names allow you to designate without supplying a pathname. They take the form "<*special name*>", where *special name* specifies text, an object, a region, or an activity, as described below.

"<SELECTION>"	References the object associated with the highlighted area, if the cursor is located in a highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the object on which the cursor is located, whether or not there is a highlighted area in the window.

"<IMAGE>"	References the highlighted object, if the cursor is in a highlighted area. If the cursor is not located in the highlighted area, this special name references the image on which the cursor is located.
"<TEXT>"	References the object named in the highlighted text in the image in the window.
"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, the special name references that activity rather than the default activity.

You can replace special names with other types of naming expressions, as accepted by that parameter.

procedure Append

```
procedure Append (Source : Name := "";
                 Target : Name := "<SELECTION>");
```

Description

Appends the source file to the target file.

This procedure copies the contents of the source file or files onto the end of the original target file.

Parameters

`Source : Name := "";`

Specifies the file or files to append to the target file. The source file is not changed. This parameter can use wildcards and special names to specify a set of files. The default null string resolves to the current library image, so it should be replaced.

`Target : Name := "<SELECTION>";`

Specifies the file to which the source file is to be appended. This parameter can use substitution characters and special names. If the file does not exist, it is created. The default is the currently selected object.

Errors

Error messages are sent to the log file.

Common errors include specification of a file that does not exist.

Example

The command:

```
file_utilities.append ("file_1","file_2");
```

takes the contents of File_1 and appends them to the end of File_2. The original contents of File_2 remain at the beginning of the file, with the contents of File_2 appended to the end. The contents of File_1 are not modified.

procedure Compare

```
procedure Compare (File_1       : Name    := "<REGION>";
                   File_2       : Name    := "<IMAGE>";
                   Subobjects   : Boolean := False);
                   Ignore_Case  : Boolean := False);
                   Options      : String  := "");
```

Description

Finds the first difference between two objects.

This procedure displays a message in the current output window when the first difference between the two files is discovered. Comparison of the two files then terminates. If the two files are identical, a message to that effect is displayed in the current output window.

The Options parameter permits pattern matching in File_2. See the introduction to this package for information on pattern matching. Certain characters in the file are interpreted as wildcards if the "File_2_Has_Wildcards" option is used. These special wildcard characters are defined as follows:

? Matches any single character.

% Matches any single character that is legal in an Ada identifier.

$ Matches the following characters, frequently used as Ada delimiters: & ' ( ) * + , - . / : ; < = > |

When not at the end of the pattern, causes the character immediately following this wildcard to be interpreted as a normal (not a wildcard) character.

{ When at the beginning of the pattern, requires the pattern to match the beginning of the line.

} When at the end of the pattern, requires the pattern to match the end of the line.

~ Matches anything except the character following this wildcard. If used inside brackets ([]), this wildcard must be the first character in the list.

* Matches zero or more occurrences of the previous character or set of characters.

[] Used around a set of characters, matches any one of the enclosed characters. Each character to be compared must be specified explicitly (for example, [ABCDE]) or by a range (for example, A-Z).

Parameters

```
File_1 : Name := "<REGION>";
```

Specifies the first file to be compared. This parameter can use special names and wildcards to specify a set of files. The default is the currently selected region.

```
File_2 : Name := "<IMAGE>";
```

Specifies the second file to be compared. In effect, this parameter acts as a template to compare File_1 against when using the "File_2_Has_Wildcards" option. This parameter can use special names. The default is the current image. The wildcards in File_2 will be interpreted as wildcards if the "File_2_Has_Wildcards" option is used.

```
Subobjects : Boolean := False;
```

Specifies whether to include any subobjects of the two named files or objects in the comparison. The default is not to compare any subobjects.

```
Ignore_Case : Boolean := False;
```

Specifies whether to do case comparison in the two files. The default is false, in which case the words "Package" and "package" would be considered different. By changing the default to true, they would not be considered different.

```
Options : String := "";
```

Specifies the options to be used.

The "File_2_Has_Wildcards" option allows pattern matching. File_2 may contain the wildcards described in "Description," above.

The "Ignore_Blank_Lines" option allows matching even though one file may contain blank lines and the other does not.

These two options cannot be used together. The default null string specifies that no options will be used.

Errors

Error messages appear in the log file.

Common errors include specification of a file that does not exist.

Example

The command:

```
file_utilities.compare ("file_1","file_2");
```

produces the following output:

```
------------------------------------------------------------------
!USERS.RJB % FILE_UTILITIES.COMPARE          STARTED 09:10:27 PM
------------------------------------------------------------------

!USERS.RJB.FILE_1 and !USERS.RJB.FILE_2 differ at line 9, byte 349.
```

References

procedure Difference

---

constant Current_Output

```
Current_Output : constant Name := "";
```

Description

Defines a constant to represent the current output file.

Typically, this file is an output window to which all output from a variety of commands in the Environment is appended.

procedure Difference

```
procedure Difference (File_1              : Name    := "<REGION>";
                      File_2              : Name    := "<IMAGE>";
                      Result              : Name    := "";
                      Compressed_Output   : Boolean := False;
                      Subobjects          : Boolean := False);
```

Description

Reports differences between two versions of an object.

If the Compressed_Output parameter is true, the procedure omits lines that are the same in both files. Noncompressed output shows every line of the two files. By default, the output is displayed to Current_Output. The output can be redirected to any file.

If the Compressed_Output parameter is false (the default), every line from each file appears in the output. Each line of the output file has a character in column 1 that indicates the origin of the line. A line that is identical in both files has a space in column 1. A line that appears only in the first file has a 1 in column 1. Likewise, a file that appears only in the second file has a 2 in column 1. For legibility, all lines have a space in column 2 regardless of origin. The line from the input file begins in column 3 of the output file.

The lines appear in the output file in the same order in which they appear in the input files. Thus, if every line in the output file that begins with a 1 (or a 2) is deleted, and then the first two columns of every line are deleted, the result is a copy of the second (or first) input file. The Strip procedure defined in this package is useful for these kinds of manipulations.

If the Compressed_Output parameter is true, lines common to both files are omitted from the output file. In this case, the output file contains a list of instructions for converting the first file to the second file. Each instruction begins with an asterisk in column 1. There are three kinds of instructions:

- *Insertions* specify a line number in the first file and a set of lines from the second file to be inserted after that line in the first file.
- *Deletions* specify a series of line numbers in the first file to be deleted.
- *Changes* specify a series of line numbers and a set of lines from the first file to be changed to a set of lines from the second file. The two sets of lines are separated by dashes.

The line numbers used when the Compressed_Output parameter is true refer to the original files. They do not reflect changes in numbering that may be caused by preceding instructions.

Parameters

```
File_1 : Name := "<REGION>";
```

Specifies the name of the first file to be compared. This parameter can use a special name or it can use wildcards to specify a set of files. The default is the currently selected region.

```
File_2 : Name := "<IMAGE>";
```

Specifies the name of the second file to be compared. This parameter can use a special name or substitution characters. The default is the current image.

```
Result : Name := "";
```

Specifies the file to which to direct the output. The default is to write output to Current_Output. If the Result parameter does not specify Current_Output, error messages are not included. If there are errors, the contents of the Result file may be misleading. To redirect both error messages and normal output, let the Result parameter default to Current_Output and use the !Commands.Log.Set_Output procedure to redirect both the log file and Current_Output before calling the Difference procedure.

```
Compressed_Output : Boolean := False;
```

Specifies the use of the compressed form of output. The default is not to use the compressed form.

```
Subobjects : Boolean := False;
```

Specifies whether to include any subobjects of the two named objects in the comparison. The default is not to compare any subobjects.

Errors

Error messages appear in the log file.

Common errors include specification of a file that does not exist.

If the Result parameter does not specify Current_Output, error messages will be separate from the output. If error messages are produced, the information in the output file specified by the Result parameter may be misleading.

Example 1

The following is an example of uncompressed output:

```
    This line is common to both files.
1   This line appeared only in the first file.
2   This line appeared only in the second file.
    This line appears in both.
    All lines from both files appear in the output
    in one of these three forms.
```

Example 2

The following is an example of compressed output:

```
* Insert after 12
2 The lines that are inserted
2 come after the instruction.

* Delete 118 .. 119
1 The lines that are deleted
1 come after the instruction.

* Change 250 .. 251
1 Both the original lines and
1 the new lines are shown
-----
2 separated by dashes.
```

References

procedure Compare

function Equal

SJM, procedure Log.Set_Output

---

procedure Dump

```
procedure Dump (File        : Name     := "<SELECTION>";
                Page_Number : Natural  := 0;
                Word_Number : Natural  := 0;
                Word_Count  : Positive := 64);
```

Description

Displays a hexadecimal dump of the named or selected file.

This procedure displays the specified number of 128-bit words beginning at the specified word number (0 through 3F hexadecimal per page) in the specified page of the file. Each word is displayed as eight 16-bit sections. The address of the word is displayed to the left of the word. The ASCII equivalent of that word is displayed in the righthand column. Nonprintable characters are displayed as vertical bars (|).

Parameters

`File : Name := "<SELECTION>";`

Specifies the file to be displayed. This parameter can use a special name. The default is the current selection. If the name specifies an object other than a file, an error occurs.

`Page_Number : Natural := 0;`

Specifies the first page to be displayed. Pages are numbered from 0 and each page consists of 64 (40 hex) 128-bit words. The default is page 0 (the beginning of the file).

`Word_Number : Natural := 0;`

Specifies the first word to be displayed. The default is word 0 (the first word of the page).

`Word_Count : Positive := 64;`

Specifies the number of words to be displayed. The default is 64 words or one page.

Errors

Errors can occur if the file does not end on a byte boundary. An error occurs only when an attempt is made to display that last partial byte.

Error messages are sent to the log file.

Example

The command:

```
file_utilities.dump("$attributes",1,16#2b#,4);
```

produces the following display in the current output window:

```
-------------------------------------------------------------
!USERS.RJB % FILE_UTILITIES.DUMP              STARTED 01:21:18 PM
-------------------------------------------------------------

Page 1
  2B0  2020 203A 2054 5255 450A 5245 434F 5645      : TRUE|RECOVE
  2C0  5259 5F4C 4F43 414C 4954 5920 2020 203A  RY_LOCALITY     |
  2D0  2020 3132 0A44 4546 4155 4C54 5F45 4C49    12|DEFAULT_ELI
  2E0  5349 4F4E 2020 2020 2020 3A20 5452 5545  SION       | TRUE
```

---

function Equal

```
function Equal (File_1       : Name    := "<REGION>";
                File_2       : Name    := "<IMAGE>";
                Subobjects   : Boolean := False;
                Ignore_Case  : Boolean := False;
                Options      : String  := "") return Boolean;
```

Description

Indicates whether the two files are identical.

This function compares the two files and determines whether they are textually equivalent.

The Options parameter permits pattern matching in File_2. (See the introduction to this package for further information on pattern matching.) Certain characters in the file are interpreted as wildcards if the "File_2_Has_Wildcards" option is used. These special characters are defined as follows:

? Matches any single character.

% Matches any single character that is legal in an Ada identifier.

$ Matches the following characters, frequently used as Ada delimiters: & ' ( ) * + , - : ; / < = > |

\ When not at the end of the pattern, causes the character immediately following this wildcard to be interpreted as a normal (not a wildcard) character.

{ When at the beginning of the pattern, requires the pattern to match the beginning of the line.

} When at the end of the pattern, requires the pattern to match the end of the line.

~ Matches anything except the character following this wildcard. If used inside brackets ([]), this wildcard must be the first character in the list.

* Matches zero or more occurrences of the previous character or set of characters.

[] Used around a set of characters, matches any one of the enclosed characters. Each character to be compared must be specified explicitly (for example, [ABCDE]) or by a range (for example, [A-Z]).

Parameters

```
File_1 : Name := "<REGION>";
```

Specifies the first file to compare. This parameter can use a special name or it can use wildcards to specify a set of files. The default is the currently selected region.

```
File_2 : Name := "<IMAGE>";
```

Specifies the second file to compare. This parameter can use a special name. The default is the current image.

```
Subobjects : Boolean := False;
```

Specifies whether to include any subobjects of the two named files or objects in the comparison. The default is not to compare any subobjects.

```
Ignore_Case : Boolean := False;
```

Specifies whether to do case comparison in the two files. The default is false, in which case the words "Package" and "package" would be considered different. By changing the default to true, they would be considered identical.

```
Options : String := "";
```

Specifies the options to be used.

The "File_2_Has_Wildcards" option allows pattern matching. File_2 may contain the wildcards described in "Description," above.

The "Ignore_Blank_Lines" option allows matching even though one file may contain blank lines and the other does not.

These two options cannot be used together. The default null string specifies that no options will be used.

```
return Boolean;
```

Returns true if the two files are identical. If sets of files are specified with wildcards, then each pair of files must be identical to return true. Otherwise, the function returns false.

Errors

Error messages appear in the log file.

Common errors include specification of a file that does not exist or specification of an invalid option.

Example

Consider the following sample section of a procedure:

```
declare
    Is_Equal : Boolean := False;
begin
    Is_Equal := File_Utilities.Equal
                    (File_1 => "foo", File_2 => "bar",
                     Subobjects => False, Ignore_Case => True,
                     Options => "Ignore_Blank_Lines");
```

The Is_Equal variable will be set to true if files Foo and Bar are identical, with the following two qualifications: case will not be considered as a factor in the match, and blank lines will be ignored in making the match.

References

procedure Compare

---

procedure Find

```
procedure Find (Pattern     : String  := "";
                File        : Name    := "<IMAGE>";
                Wildcards   : Boolean := False;
                Ignore_Case : Boolean := True;
                Result      : Name    := "");
```

Description

Displays each line of the file that contains a match of the pattern.

This procedure searches the specified file for the pattern. (See the introduction to this package for further information on pattern matching.) The pattern can consist of any string. Certain characters in the string are interpreted as wildcards if the Wildcards parameter is true. These special characters are defined as follows:

? Matches any single character.

% Matches any single character that is legal in an Ada identifier.

$ Matches the following characters, frequently used as Ada delimiters: & ' ( ) * + , - . : ; < = > |

\ When not at the end of the pattern, causes the character immediately following this wildcard to be interpreted as a normal (not a wildcard) character.

{ When at the beginning of the pattern, requires the pattern to match the beginning of the line.

} When at the end of the pattern, requires the pattern to match the end of the line.

~ Matches anything except the character following this wildcard. If used inside brackets ([]), this wildcard must be the first character in the list.

* Matches zero or more occurrences of the previous character or set of characters.

[] Used around a set of characters, matches any one of the enclosed characters. Each character to be compared must be specified explicitly (for example, [ABCDE]) or by a range (for example, [A-Z]).

Parameters

```
Pattern : String := "";
```

Specifies the pattern for which to search.

```
File : Name := "<IMAGE>";
```

Specifies the file in which to search for the pattern. This parameter can use a special name or it can use wildcards to specify a set of files. The default is the current image.

```
Wildcards : Boolean := False;
```

Specifies whether to interpret certain characters in the Pattern parameter as wildcard characters. The default is not to interpret those characters as wildcard characters.

```
Ignore_Case : Boolean := True;
```

Specifies whether to do case comparison in the two files. The default is true, in which case the words "Package" and "package" would be considered identical. By changing the default to false, they would not be considered identical.

```
Result : Name := "";
```

Specifies the file to which to direct the output. The default is to write output to the Current_Output. If the Result parameter does not specify Current_Output, error messages are not included. If there are errors, the contents of the Result file may be misleading. To redirect both error messages and normal output, let the Result parameter default to Current_Output and use the !Commands.Log.Set_Output procedure to redirect both the log file and Current_Output before calling the Find procedure.

Errors

Error messages are sent to the log file.

A common error is incorrect specification of the pattern for which to search.

Example 1

The command:

```
find ("test","section 3");
```

finds the string "Section 3" in the specified file, Test.

Example 2

The command:

```
find ("test","[^abc]",true);
```

matches anything except the characters A, B, and C in the specified file.

References

function Found

SJM, procedure Log.Set_Output

---

function Found

```
function Found (Pattern     : String  := "";
                File        : Name    := "<IMAGE>";
                Wildcards   : Boolean := False;
                Ignore_Case : Boolean := True) return Natural;
```

Description

Finds the number of lines that contain matches of the pattern in the file.

This function searches the specified file for the pattern. The pattern can consist of any string. (See the introduction to this package for further information on pattern matching.) Certain characters in the string are interpreted as wildcards if the Wildcards parameter is true. These special characters are defined as follows:

? Matches any single character.

% Matches any single character that is legal in an Ada identifier.

$ Matches the following characters, frequently used as Ada delimiters: & ' ( ) * + , - . : ; / < = > |

\ When not at the end of the pattern, causes the character immediately following this wildcard to be interpreted as a normal (not a wildcard) character.

{ When at the beginning of the pattern, requires the pattern to match the beginning of the line.

} When at the end of the pattern, requires the pattern to match the end of the line.

^ Matches anything except the character following this wildcard. If used inside brackets ([]), this wildcard must be the first character in the list.

* Matches zero or more occurrences of the previous item.

[] Used around a string of characters, matches any one of the enclosed characters. Each character to be compared must be specified explicitly (for example, [ABCDE]) or by a range (for example, [A-Z]).

Parameters

```
Pattern : String := "";
```

Specifies the pattern for which to search.

```
File : Name := "<IMAGE>";
```

Specifies the file in which to search for the pattern. This parameter can use a special name or it can use wildcards to specify a set of files. The default is the current image of the file.

```
Wildcards : Boolean := False;
```

Specifies whether to interpret certain characters as wildcard characters. The default is not to interpret those characters as wildcard characters.

```
Ignore_Case : Boolean := True;
```

Specifies whether to do case comparison in the two files. The default is true, in which case the words "Package" and "package" would be considered identical. By changing the default to false, they would be considered different.

```
return Natural;
```

Returns the number of lines in the file that contain matches of the pattern.

Errors

Error messages are sent to the log file.

A common error is incorrect specification of the pattern.

Example 1

The function:

```
found ("test","section 3");
```

returns the number of lines that contain the string "Section 3" in the specified file.

Example 2

The function:

```
find ("test","[^abc]",true);
```

returns the number of lines that contain anything except the characters A, B, and C.

References

procedure Find

procedure Merge

```
procedure Merge (Original  : Name := "";
                 File_1    : Name := "";
                 File_2    : Name := "";
                 Result    : Name := "");
```

Description

Merges two variants of the same object into a new object.

Each variant is compared separately with the original file. This procedure then tries to accommodate all nonconflicting changes in the two variants. If it encounters variations it cannot reconcile, the procedure indicates this in the Message window and marks conflicting differences in the same format used by the Difference procedure.

Parameters

`Original : Name := "";`

Specifies the name of the file against which the two variants are compared. This parameter can use a special name or it can use wildcards to specify a set of files.

`File_1 : Name := "";`

Specifies the name of the first file to be merged. This parameter can use a special name or it can use substitution characters or special names to create filenames from the Original parameter.

`File_2 : Name := "";`

Specifies the name of the second file to be merged. This parameter can use a special name or it can use substitution characters or special names to create filenames from the Original parameter.

`Result : Name := "";`

Specifies the file to which to write the result. The default is to write output to the current output window.

Errors

Error messages are sent to the log file.

The most common error is incorrect specification of a filename.

Example

The command:

```
file_utilities.merge ("original","foo","bar","result");
```

merges files Foo and Bar, compares them to file Original, and places the results in file Result.

References

procedure Strip

---

subtype Name

```
subtype Name is String;
```

Description

Defines the names of objects.

This subtype allows the use of special names, wildcards in the Source parameter, context prefixes, and attributes that are defined for general naming. See the Key Concepts in this book for more information about wildcards, context prefixes, special names, and attributes.

---

procedure Strip

```
procedure Strip (Source : Name := "<SELECTION>";
                 Target : Name := "");
```

Description

Takes the output of the Merge or the Difference procedure and creates a clean file.

This procedure removes the annotations inserted into the source file by the Merge procedure or the Difference procedure. The Strip procedure is useful for taking the result of Merge or Difference and removing the variation notations that those procedures place in column 1 of the result.

The user can edit the output to resolve conflicting changes before stripping.

Parameters

`Source : Name := "<SELECTION">;`

Specifies the source file to be stripped. This file is not changed. This parameter can use a special name or it can use wildcards to specify a set of files. The default is the current selection.

`Target : Name := "";`

Specifies the file into which the stripped source is to be placed. If this file does not already exist, it is created. If it does exist, its previous contents are lost. This parameter can use special names. The default is to display the results in the current output window.

Errors

Error messages are sent to the log file.

The most common error is incorrect specification of a filename.

Example

The following example illustrates the results of using the Strip procedure on a file created by the Difference procedure.

The procedure:

```
file_utilities.strip (source=>"text3",target=>"text4");
```

was run on the file Text3 below:

```
* Object 1: !USERS.GZC.WM_FILE_UTILITIES.TEXT1
* Object 2: !USERS.GZC.WM_FILE_UTILITIES.TEXT2
1 This is text from file one.
2 This is text from file two.
  This is text that is in both files.
1 This is more text from file one.
2 This is more text from file two.
```

The results placed in file Text4 are:

```
Object 1: !USERS.GZC.WM_FILE_UTILITIES.TEXT1
Object 2: !USERS.GZC.WM_FILE_UTILITIES.TEXT2
This is text from file one.
This is text from file two.
This is text that is in both files.
This is more text from file one.
This is more text from file two.
```

References

procedure Merge

---

end File_Utilities;

---

package Library

Package Library provides commands for manipulating objects in the library system and providing type-specific editing for library images. As with most commands in the world !Commands, these commands can be executed from programs but are tailored for use as interactive commands.

Access Control and Library Commands

Access to worlds, Ada units, and files is controlled by the access lists (ACLs) associated with each object of these types. Thus, when you perform operations on objects to which you do not have the required access, error messages will be generated indicating that you do not have the required access class. For further information on access control, see "Access Control" in the Key Concepts in this book.

Error Response

The commands in this package have a Response parameter that specifies how the command should respond to errors, how to generate logs, and what activities to use. The response profile "<PROFILE>", which many commands use by default, specifies the job response profile. If there is no job response profile, the session response profile ("<SESSION_PROFILE>") is used. If there is no session response profile, the system's default profile ("<DEFAULT>") is used. For further information on profiles, see SJM, package Profile.

Many of the commands in package !Commands.Common also apply to library images. The applicable procedures from that package are described below.

The common editing operations are discussed more fully in EST, package Common.

Image Structure

Here is an example of the image of a library for the home library for user Lance:

```
!Users.Lance
  A_Generic_Instantiation
  A_Generic_Package
  A_Generic_Package
  A_Package
  A_Package
    .Nested
    .T
  A_Procedure
  A_Procedure
    ._Ada_1_
  Control_Link
  Current_Release
  File_From_Direct_Io
  File_From_Text_Io
  Library_Switches
  Nested_Directory
  Nested_World
  S_1
  S_1_Switches
  _Ada_7_
```

By default, this image displays the fully qualified name of the library on the first line of the image. The rest of the lines of the image are names of the objects in the library, sorted in alphabetical order.

Note that subunits of Ada units are displayed on the lines following their parents, prefixed with a period (.). Insertion points or withdrawn items are also listed. Their names begin with the characters _Ada_.

The banner for library images (it appears under the window displaying the image) indicates the name of the library, an image type of library, and some additional information, including:

- Whether the library is frozen.
- The name of the target key for the library if it is not R1000.
- Whether it is a world or a directory.
- The current elision level (see "Elision and Expansion," below).

You can change the display to show more detail by using the !Commands.Common-.Explain command. Displaying this additional detail means that it takes longer to obtain the images of new libraries or to update changed library images. The default display for the standard additional information available for the same library is:

```
!Users.Lance : Library (World);
  A_Generic_Instantiation : Ada (Pack_Inst);
  A_Generic_Package       : Ada (Gen_Pack);
  A_Generic_Package       : Ada (Pack_Body);
  A_Package               : Ada (Pack_Spec);
  A_Package               : Ada (Pack_Body);
    .Nested               : Ada (Pack_Body);
    .T                    : Ada (Task_Body);
  A_Procedure             : Ada (Proc_Spec);
```

```
A_Procedure              : Ada (Proc_Body);
  ._Ada_1_               : Ada (Statement);
Control_Link             : Pipe;
Current_Release          : File (Activity);
File_From_Direct_Io      : File (Binary);
File_From_Text_Io        : File (Text);
Library_Switches         : File (Switch);
Nested_Directory         : Library (Directory);
Nested_World             : Library (World);
S_1                      : Session;
S_1_Switches             : File (Switch);
_Ada_7_                  : Ada (Comp_Unit);
```

The additional information includes the class of the object and the subclass of the object enclosed in parentheses. The subclass of the object can be helpful in distinguishing the different kinds of Ada units (specs, bodies, instantiations, and so on) or different kinds of files (text, binary, activity, and so on). A list of classes and subclasses is included in the Key Concepts in this book.

Even more miscellaneous information is available by default by using the !Commands.Common.Explain command. This information would be displayed as follows for the above library:

```
!Users.Lance : Vol: 4  (1);
  A_Generic_Instantiation : I 86/06/02 18:57:19 Lance     5251  ( 1) Frz;
  A_Generic_Package       : I 86/06/02 18:56:24 Lance     2968  ( 1)     ;
  A_Generic_Package       : I 86/06/02 18:56:37 Lance     2934  ( 1)     ;
  A_Package               : I 86/06/02 18:51:42 Lance     2885  ( 1)     ;
  A_Package               : I 86/06/02 18:53:51 Lance     6261  ( 1)     ;
    .Nested               : I 86/06/02 18:52:53 Lance     3052  ( 1)     ;
    .T                    : I 86/06/02 18:53:58 Lance     3047  ( 1)     ;
  A_Procedure             : C 86/06/02 18:54:51 Lance     2880  ( 1)     ;
  A_Procedure             : C 86/06/02 19:03:14 Lance     3161  ( 1)     ;
    ._Ada_1_              : S 86/06/02 19:03:15 Lance      665  ( 1)     ;
  Control_Link            :   86/06/02 18:51:03 Lance        0  ( 1)     ;
  Current_Release         :   86/06/02 18:58:33 Lance      187  ( 1) Frz;
  File_From_Direct_Io     :   86/06/02 18:48:28 Lance       17  ( 1)     ;
  File_From_Text_Io       :   86/06/02 18:48:27 Lance       17  ( 1)     ;
  Library_Switches        :   86/06/02 19:00:11 Lance       45  ( 1)     ;
  Nested_Directory        :                     Vol 4           ( 1)     ;
  Nested_World            :                     Vol 4           ( 1)     ;
  S_1                     :   86/06/03 18:42:59 *System      0  ( 1)     ;
  S_1_Switches            :   86/06/03 18:36:14 Lance      278  ( 1)     ;
  _Ada_7_                 : S 86/06/02 19:01:50 Lance      651  ( 1)     ;
```

This information includes (from left to right):

- The unit state of the object if it is an Ada unit: archived (A), source (S), installed (I), or coded (C).
- The date and time of the last edit of the object.
- The user who last edited the object or the volume number if it is a library.
- The size of the object in bytes.
- The number of versions to be retained for the object (the retention count).

- An indication if the object is frozen.

In addition to the detailed information available for each object, information on the deleted objects and retained versions of objects is available. For example, using the !Command.Common.Expand command results in the following display:

```
!Users.Lance
  {A_Deleted_Unit'V(3)}
  A_Generic_Instantiation'V(3)
- A_Generic_Instantiation'V(2)
  A_Generic_Package'V(3)
- A_Generic_Package'V(2)

  A_Generic_Package'V(3)
- A_Generic_Package'V(2)
  A_Package'V(3)
- A_Package'V(2)
  A_Package'V(4)
  ...
```

Note that deleted objects are enclosed in braces ({}). Versions of the objects are names using version qualification. The nondefault versions of objects are prefixed with the hyphen (-).

All of the above information can be obtained by using the !Commands.Common.Expand command, the !Commands.Common.Explain command, or session switches. For more information, see "Elision and Expansion" and "Session Switches," below.

Key Concepts

Designation

Designation includes both selection and cursor position to indicate an object on which to operate.

Selections can be made using either the selection commands from package !Commands.Common.Object or the region selection commands from package !Commands-.Editor.Region, most of which are bound to keys.

Special Names

Special names are used as parameter values for many Environment operations to specify text, objects, and regions. They take the form "<*special name*>", where *special name* specifies the text, object, region, or activity that they represent. A special name can be used anywhere that a string name can be used.

The following list shows the special names used in the Environment and what they reference:

"<SELECTION>"	References the object associated with the highlighted area, if the cursor is located in a highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the highlighted object on which the cursor is located, whether or not there is a highlighted area in the window.
"<IMAGE>"	References the highlighted object, if the cursor is in a highlighted area. If the cursor is not located in the highlighted area, this special name references the image on which the cursor is located.
"<TEXT>"	References the object named in the highlighted text in the image in the window.
"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, this special name references that activity rather than the default activity.

You can replace special names with other types of naming expressions, as accepted by that parameter.

Special Values

The operations in this package can use several special values when specifying a parameter of the Compilation.Change_Limit type. Parameters of the Compilation-.Change_Limit type control which units an operation can modify, as follows:

"<SUBUNITS>"	Modifies only the units named in the operation and their subunits.
"<UNITS>"	Modifies only the units named in the operation.
"<DIRECTORIES>"	Modifies only the units in the same set of directories as the units specified to the operation.
"<WORLDS>"	Modifies only the units in the same world as the units specified to the operation.
"<ALL_WORLDS>"	Modifies a unit in any world.

Parameter Placeholders

Many of the commands in this package have, as a default parameter value, a parameter placeholder of the form "*>>name <<*", where *name* is the type of object that should replace *>>name <<*. Parameter placeholders must be replaced by the name of an object, as specified by their type. Executing a command containing a parameter placeholder will result in an error.

Elision and Expansion

The two different types of detail that can be expanded and elided are described below. Displaying additional detail means that it takes longer to obtain the images of new libraries or to update changed library images.

Additional information on the objects, versions, and deleted objects in the library can be added and removed with the Expand and Elide commands from package !Commands.Common. Table 7–1 describes the elision levels available. They are arranged in order with the most expanded level at the top and the most elided level at the bottom. Note that the default elision level is blank (that is, there is nothing on the banner).

Table 7–1. Elision Levels

Banner Symbol	*Descriptions*
(versions)	All deleted and undeleted versions
versions	All undeleted versions
(lib vers)	All deleted and undeleted versions; no subunits
lib vers	All undeleted versions; no subunits
(units)	All deleted and undeleted objects
units	All undeleted objects
(blank)	Default versions of objects and subunits
(lib units)	All deleted and undeleted objects; no subunits
lib units	All undeleted objects; no subunits

Additional information pertaining to specific objects can be obtained by designating an object or set of objects and executing the !Commands.Common.Explain command to cycle between the default, standard, and miscellaneous display categories. The information displayed for each of these categories is determined by the value of session switches (see "Session Switches," below, for more information on session switches). The first three examples of library images in "Image Structure," above, presented the default information made available in the default, standard, and miscellaneous displays, respectively.

Session Switches

Many session switches determine how library images are displayed. See SJM, Session Switches, for more information on session switches.

The following session switches pertain to libraries:

Library_Break_Long_Lines(default true)

Controls whether lines that exceed the value of the Library_Line_Length session switch are broken.

Library_Capitalize (default true)

Determines whether identifiers in library images are capitalized.

Library_Indentation (default 2)

Determines how much to indent subunit names when displayed in the short form (that is, without their parent name as prefixes, which is determined by the value of the Library_Shorten_Names session switch).

Library_Lasy_Realignment (default true)

Determines whether the image is realigned when a longer name is added. If true, the Environment waits for a Redraw request.

Library_Line_Length (default 80)

Determines how long a line can be before it is eligible to be broken.

Library_Misc_Show_Edit_Info (default true)

Shows time and user of last update/edit for the version when displaying miscellaneous information.

Library_Misc_Show_Frozen (default true)

Shows "Frz" for frozen objects when displaying miscellaneous information.

Library_Misc_Show_Retention (default true)

Shows the retention count (for example, 10) when displaying miscellaneous information.

Library_Misc_Show_Size (default true)

Shows the size of the version in bytes when displaying miscellaneous information.

Library_Misc_Show_Subclass (default false)

Shows the object's subclass when displaying miscellaneous information.

Library_Misc_Show_Unit_State (default true)

Shows the shortened form of the unit state of Ada objects when displaying miscellaneous information.

Library_Misc_Show_Volume (default true)

Shows the volume for libraries when displaying miscellaneous information.

Library_Shorten_Names (default true)

Determines whether the pathname of the parent is displayed for subunits.

Library_Shorten_Subclass (default true)

Shows subclasses in shortened form when displaying miscellaneous information.

Library_Shorten_Unit_State (default true)

Shows only the first letter of the unit state when displaying miscellaneous information.

Library_Show_Deleted_Objects (default false)

Shows deleted objects (for example, {Foo'Body}) when displaying miscellaneous information; controlled by elision.

Library_Show_Deleted_Versions (default false)

Shows version numbers and information for all versions of an object when displaying miscellaneous information; controlled by elision.

Library_Show_Miscellaneous (default false)

Shows miscellaneous information; obtainable by using the !Commands.Common.Explain command; controlled by elision.

Library_Show_Standard (default false)

Shows standard information; obtainable by using the !Commands.Common.Explain command; controlled by elision.

Library_Show_Subunits (default true)

Shows subunits in the initial display when displaying miscellaneous information; controlled by elision.

Library_Show_Version_Number (default false)

Shows the version number of the default or maximum version as part of the object name (for example, Foo'V(4) or Bar'V(2)) when displaying miscellaneous information.

Library_Std_Show_Class (default true)

Shows class along with subclass—for example, File (Text) instead of Text when displaying miscellaneous information.

Library_Std_Show_Subclass (default true)

Shows subclass as part of the standard information.

Library_Std_Show_Unit_State (default false)

Shows unit state for Ada units as part of the standard information display.

Library_Uppercase (default false)

Determines whether identifiers in library images are uppercased. Many users prefer a default library image that appears as follows:

```
!Users.Lance :   Library (World);
  A_Generic_Instantiation  : I Ada (Pack_Inst);
  A_Generic_Package        : I Ada (Gen_Pack);
  A_Generic_Package        : I Ada (Pack_Body);
  A_Package                : I Ada (Pack_Spec);
  A_Package                : I Ada (Pack_Body);
    .Nested                : I Ada (Pack_Body);
    .T                     : I Ada (Task_Body);
  A_Procedure              : C Ada (Proc_Spec);
  A_Procedure              : C Ada (Proc_Body);
    ._Ada_1_               : S Ada (Statement);
  Control_Link             :   Pipe;
  Current_Release          :   File (Activity);
  File_From_Direct_Io      :   File (Binary);
  File_From_Text_Io        :   File (Text);
  ...
```

To establish this as your default libraries display, modify the following session switch values:

- Library_Show_Standard := True
- Library_Std_Show_Unit_State := True

This causes the standard additional information to be displayed by default and unit state information to be added to this standard information.

Commands from package Common

The following commands from package !Commands.Common are supported for editing libraries:

procedure Common.Abandon

Ends the editing of the specified image. The window is removed from the screen and from the Window Directory. The Window parameter allows you to specify which window should be removed. The default is the current image, unless there is a selection in that image. In that case, the selection is abandoned. Procedure Common.Complete refreshes the library image in the current window to the current value of the underlying permanent representation and realigns the columns of the image.

procedure Common.Complete

Refreshes the library image in the current window to the current value of the underlying permanent representation and realigns the columns of the image.

procedure Common.Create_Command

Creates a Command window below the current library window if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the current library window. This Command window initially has a *use* clause:

```
use Editor, Library, Common;
```

This *use* clause provides direct visibility to the declarations in packages Editor, Library, and Common without requiring qualification for names resolved in the command.

procedure Common.Definition

Finds the defining occurrence of the named or designated element and brings up its image in a window on the screen. If a name is provided, it is used. If no name is provided, a selection with the cursor in it is used if one exists. Otherwise, the cursor location is used to designate the element. An In_Place parameter specifies whether the existing window should be used. A Visible parameter specifies whether to go to the visible part or the body (if possible).

procedure Common.Demote

Demotes the selected Ada unit to the next lower state. The procedure changes the state of the selected Ada unit, assuming there are no other units dependent on the unit. If there are dependent units, a list of them is displayed in the menu window that is brought onto the screen. See EST, Menus, for more information on the editing operations available on menus.

The specific effect of this procedure depends on the current state of the unit. If the current state is:

- Archived: The procedure has no effect.
- Source: The procedure has no effect.
- Installed: The unit is demoted to the source state.
- Coded: The unit is demoted to the installed state.

procedure Common.Edit

Creates a window in which to edit the named or selected object. An In_Place parameter specifies whether the existing window should be used. A Visible parameter specifies whether to bring up the visible part or the body (if possible).

For more information on the actions performed, see EST, procedure Common.Edit, for the class of object being edited.

procedure Common.Elide

Reduces the level of detail displayed for the designated object(s). See "Elision and Expansion," above, for more information.

procedure Common.Enclosing

Finds the parent library unit of the current library and displays that parent in a window. An In_Place parameter specifies whether the existing window should be used. A Library parameter specifies whether the resulting image should be a library rather than the parent body when the parent body is not a library.

procedure Common.Expand

Increases the level of detail displayed for the designated object(s). See "Elision and Expansion," above, for more information.

procedure Common.Explain

Changes the level of detail displayed for the designated object(s) in the library. There are three levels:

- Default information
- Standard information
- Miscellaneous information

This command cycles through the levels, proceeding down the list and cycling back to the top when at the bottom. See "Elision and Expansion," above, for more information.

procedure Common.Format

Refreshes the library image in the current window to the current value of the underlying permanent representation and realigns the columns of the image.

This performs the same operation as the Common.Revert procedure.

procedure Common.Promote

Promotes the selected Ada object to the next higher unit state. The specific effect of this procedure depends on the current unit state of the unit. If the current state is:

- Archived: The unit is promoted to the source state.
- Source: The unit is promoted to the installed state.
- Installed: The unit is promoted to the coded state.
- Coded: If the unit is selected, execution is attempted. If parameters are required, the prompt for them appears in a Command window.

procedure Common.Release

Ends the editing of the library image. The library image window is removed from the screen and from the Window Directory.

procedure Common.Revert

Refreshes the library image in the current window to the current value of the underlying permanent representation and realigns the columns of the image.

This performs the same operation as the Common.Format procedure.

procedure Common.Undo

Undeletes the selected object. This procedure is similar to the Library.Undelete procedure.

procedure Common.Object.Child

Selects the child of the current selection. The procedure selects the line the cursor is on if there are no selections or if the cursor is not in the selection. If there is a line selected, the procedure selects the first child of that line. If the selected line has no child, it selects the next line.

procedure Common.Object.Copy

Copies the selected object into the image where the cursor is located. The procedure prompts with a Library.Copy command in a Command window below the window in which the cursor is located. The From parameter has the name of the selected object as the default value and the To parameter has the current context as the default value.

procedure Common.Object.Delete

Deletes the selected object. If other elements are dependent on the element because of semantic references (from installed or coded units), the deletion fails, a menu of the dependent units is displayed in the menu window, and a Library.Delete command with the name of the selected unit as the parameter is placed in a Command window. For more information, see the description of the editing operations on menus in EST, Menus. Contained units of the selected element are not deleted. The cursor must be in the selection for the operation to succeed.

procedure Common.Object.First_Child

Selects the first child of the current selection. The procedure selects the line the cursor is on if there are no selections or if the cursor is not in the selection. If there is a line selected, the procedure selects the first child of that line. If the selected line has no child, it selects the next line.

procedure Common.Object.Insert

Creates an insertion point in a library where an Ada compilation unit can be inserted.

procedure Common.Object.Last_Child

Selects the last child of the current selection. If there is no selection in the image or the cursor is not in the selection, this procedure selects the current line. If there is a selection, the procedure selects the last child of the current selection. If the selection has no subobjects, it selects the next object.

procedure Common.Object.Move

Moves the selected object into the library in which the cursor is located. The procedure prompts with a Library.Move command in a Command window below the library in which the cursor is located. The From parameter specifies, as a default, the selected object, and the To parameter specifies, as a default, the library in which the cursor is located.

procedure Common.Object.Next

Selects the next object at the same or greater level past the currently selected object.

procedure Common.Object.Parent

Selects the parent of the current selection. If there is no selection or if the cursor is not in the selection, the procedure selects the line on which the cursor is located.

procedure Common.Object.Previous

Selects the previous object at the same or greater level before the currently selected object.

constant Ada_Format

```
Ada_Format : constant Fields := Fields'(Status      => True,
                                        Declaration => True,
                                        others      => False);
```

Description

Defines a constant that specifies a set of data for Ada objects to be displayed by the List procedure.

This is the default value of the Displaying parameter in the Ada_List procedure.

renamed procedure Ada_List

```
procedure Ada_List (Pattern     : Name    := "<IMAGE>@'C(ADA)";
                    Displaying  : Fields  := Library.Ada_Format;
                    Sorted_By   : Field   := Library.Declaration;
                    Descending  : Boolean := False;
                    Response    : String  := "<PROFILE>";
                    Options     : String  := "") renames List;
```

Description

Displays the specified set of data about the specified set of versions of specified objects.

This procedure performs exactly as the List procedure except that it has different default parameters. The default parameters provide a display of all Ada units in the current context.

Further explanation and examples can be found in the Key Concepts in this book.

Parameters

`Pattern : Name := "<IMAGE>@'C(ADA)";`

Defines the set of objects to be listed. Wildcards, context prefixes, and attributes can be used in this name. The default gives the set of objects of the class Ada in the current image.

`Displaying : Fields := Library.Ada_Format;`

Specifies the set of data to list about each object. The default is to list the Ada format set of data.

`Sorted_By : Field := Library.Declaration;`

Specifies the field that should be sorted to order the list. The default is to order by the declaration.

`Descending : Boolean := False;`

Specifies whether to reverse the ordering. The default is to use the natural ascending order.

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

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

```
Options :  String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

Example

Consider the following world:

```
!Users.Gzc           :   Library (World);
  File_1             :   File (Text);
  Library            :   Library (Directory);
  My_Unit            : | Ada (Pack_Spec);
  My_Unit            : | Ada (Pack_Body);
  Sample_Directory   :   Library (Directory);
  Sample_World       :   Library (World);
  S_1                :   Session;
  S_1_Switches       :   File (Switch);
```

The command:

```
ada_list ("!users.gzc.@'c(ada)");
```

displays all of the Ada units within that world, their status, and their subclass, as follows:

```
------------------------------------------------------------------------------

!USERS.GZC % ADA_LIST                                      STARTED 2:48:09 PM
------------------------------------------------------------------------------


87/01/06 14:48:11 ::: Listing of !USERS.GZC.@'C(ADA) sorted by declaration.

 STATUS            DECLARATION
=========  ==========================
INSTALLED  My_Unit : Ada (Pack_Spec);
INSTALLED  My_Unit : Ada (Pack_Body);

87/01/06 14:48:12 ::: [End of Library.List command -- No errors detected].
```

---

constant All_Fields

```
All_Fields : constant Fields := Fields'(others => True);
```

Description

Defines a constant that specifies that all fields of data be displayed for the objects displayed by the List procedure.

procedure Compact_Library

```
procedure Compact_Library (Existing : Name   := "<SELECTION>";
                           Response : String := "<PROFILE>");
```

Description

Reduces the amount of storage consumed by frequently modified libraries (worlds or directories).

As objects in libraries are created and destroyed, the library accumulates space that contains unneeded information. This procedure removes this unneeded information, thus reducing the space required for use by the directory.

While the library is being compacted, no other jobs should be referencing it.

Parameters

`Existing : Name := "<SELECTION>";`

Specifies the name of the library to be compacted. The default is the current selection.

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

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

Example

The command:

```
library.compact_library ("!users.gzc");
```

compacts the specified library and results in the following display confirming the compaction:

```
!USERS.GZC % LIBRARY.COMPACT_LIBRARY                    STARTED 2:57:44 PM
-------------------------------------------------------------------------------

87/01/06 14:57:45 ::: [Library.Compact_Library ("!users.gzc", PERSEVERE);].
87/01/06 14:57:47 --- GZC's size was 9658 bytes, new size is 8237.
87/01/06 14:57:47 +++ !USERS.GZC has been compacted.
87/01/06 14:57:47 ::: [End of Library.Compact_Library command -- No errors
87/01/06 14:57:47 ... detected].
```

procedure Context

```
procedure Context (To_Be     : Context_Name := "$";
                   Response  : String       := "<PROFILE>");
```

Description

Sets the current context to the specified context.

This procedure sets the default context to the specified context. The default context is then displayed in the current log.

The context is set on a per-job basis. The initial context for any job is the library that enclosed the object displayed in the window from which the job was initiated. Note that commands executed in a Command window run as a separate job. Using this procedure in such a job changes the default context only for the duration of that job.

Parameters

`To_Be : Context_Name := "$";`

Specifies the new context. The default is the current context, which has the effect of displaying the context without changing it. Wildcards, attributes, and context prefixes can be used in this name if the name resolves unambiguously to only one location.

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

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

Example

The command:

```
library.context ("!users.gzc.sample_directory");
```

executed in a Command window, changes the context to !Users.Gzc.Sample_Directory for the duration of that command execution only.

subtype Context_Name

```
subtype Context_Name is Name;
```

Description

Defines the name of a world or directory that is the context for other commands or name resolution.

The name can use special names, wildcards, and context prefixes but must resolve to a unique directory or world. See the Key Concepts in this book for more information about names in general.

procedure Copy

```
procedure Copy (From       : Name    := "<REGION>";
                To         : Name    := "<IMAGE>";
                Recursive  : Boolean := True;
                Response   : String  := "<PROFILE>";
                Copy_Links : Boolean := True;
                Options    : String  := "");
```

Description

Copies the value of an existing object into another object.

This procedure can copy a single object or a hierarchy of objects. Wildcards can be used to specify a set of objects to be copied. The Recursive parameter allows any subobjects of the named object to be copied. If more than one object is copied, each object is copied independent of any other.

This procedure creates a new version of the existing object if the new object already exists. This may force old versions of the object to be expunged.

If the new object does not exist, it is created. For all Ada units, the new object is created in the source state. The new object created is of the same class as the object from which it was copied.

This procedure is used for the following purposes:

- To create a copy of an object with the same or a different simple name in another directory.
- To create a copy of an object with a different name in the same directory.
- To copy the links that are associated with each world. The set of links for a world do not have a name, so they are not copied as an object in a world. The Copy_Links parameter allows the procedure to copy those links.

Copying an object from one library to another with the same simple name can be accomplished by using the name of the destination library as the To parameter.

Table 7–2 illustrates the results of executing the Copy procedure with various types of objects as the To and From parameters. The To parameter objects are shown horizontally and the From parameter objects are shown vertically in the table.

The word TO indicates that the object specified by the To parameter is a copy of the object specified by the From parameter. The word INTO indicates that the object specified by the From parameter is copied into the object specified by the To parameter.

Table 7-2. Using the Copy Procedure with To and From Parameters

			To Parameter			
From Parameter	Non-Ada object	Library unit	Subunit	World	Directory	No object
Non-Ada object	TO (1)	Error	Error	INTO	INTO	TO
Library unit (2)	Error	TO	TO	INTO	INTO	TO
Subunit (2)	Error	INTO	TO	INTO	INTO	TO
World (3)	Error	Error	Error	TO (4)	TO (4)	TO
Directory (3)	Error	Error	Error	TO (4)	TO (4)	TO

The number in parentheses following the results indicates a restriction on the copy. These restrictions are:

(1) The objects must be of the same class.

(2) If Recursive is true, the subunits of the unit are involved. The relative nesting of subunits is preserved.

(3) If Recursive is true, the subcomponents of the library are involved. The relative nesting of subcomponents is preserved.

(4) The contents of the From library are merged with the contents of the To library.

Parameters

```
From : Name := "<REGION>";
```

Specifies the existing object or objects to be copied. The name can use special names, wildcards, context prefixes, and attributes. The default is the selection, whether or not the cursor is in the selection.

```
To : Name := "<IMAGE>";
```

Specifies the name of the new object. The To parameter is interpreted in the current context or specified full context and must be unique. The name can use substitution characters to create the new name from the existing name. If the named object exists, the old value of the object is deleted. If the named object does not exist, it is created. The default is the current image.

```
Recursive : Boolean := True;
```

Specifies whether to copy any contained objects. The default is to copy all contained objects.

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

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

```
Copy_Links : Boolean := True;
```

Specifies whether to copy the links that are associated with any world that is copied. The default is to copy all links. If Copy_Links is false when copying worlds, the copied worlds will have an empty set of links.

```
Options : String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

Restrictions

Objects representing devices cannot be copied.

Any situation that would require demoting unrelated declarations results in an error, suppressing the copy.

If a library and its switch file are copied, the copied library will point to the copy of the switch file. If the switch file is not copied, the library and the original from which it was copied will reference the original switch file.

For Ada units, changing the simple name during a copy may require changing the name in the Ada unit declaration before installation.

Example 1

Consider the following world:

```
!User.Rjb
---------
    Sample : File;
    Macros : Ada (Pack_Spec);
    Macros : Ada (Pack_Body);
```

The command:

```
copy ("$sample","$sample2");
```

creates a copy of the file. The world now appears as follows:

```
!User.Rjb
---------
    Sample   : File;
    Macros   : Ada (Pack_Spec);
    Macros   : Ada (Pack_Body);
    Sample2  : File;
```

Example 2

Consider the following world:

```
!User.Rjb
---------
    Sample : File;
    Macros : Ada (Pack_Spec);
    Macros : Ada (Pack_Body);
```

User Jpl would like to copy those macros into his home world.

The command:

```
copy ("$macros","!users.jpl.macros");
```

copies the macros into user Jpl's home world. The world Rjb is not changed, but package Macros and all subunits are copied into !Users.Jpl. If package Macros already exists in the world Jpl, a new version of the package is created and older versions may be lost.

Example 3

Suppose user Jpl wants to copy those macros into his home world but wants to call them New_Macros.

The command:

```
copy ("$macros","!users.jpl.new_macros");
```

accomplishes this.

References

procedure Move

---

procedure Create

```
procedure Create (Name      : Library.Name  := ">>LIBRARY NAME<<";
                  Kind      : Library.Kind  := Library.Directory;
                  Vol       : Volume        := Library.Nil;
                  Model     : String        := "!Model.R1000";
                  Response  : String        := "<PROFILE>");
```

Description

Creates a library or package in the directory system with the specified name and class on the specified disk volume.

This procedure creates a new object in the library system. The object can be a directory, a world, or a package. If the object is a world, the disk volume on which the library is built can be specified. If the object is a package or a directory, the volume on which it is built is inherited from the enclosing world.

Parameters

`Name : Library.Name := ">>LIBRARY NAME<<";`

Specifies the name of the new unit. Wildcards, context prefixes, and attributes can be used except in the last segment (the simple name) of the unit. The name must specify a single object that does not already exist. If the default parameter placeholder ">>LIBRARY NAME<<" is not replaced, an error will result.

`Kind : Library.Kind := Library.Directory;`

Specifies the kind of object to be built. The default is a directory.

`Vol : Volume := Library.Nil;`

Specifies the disk volume on which the object is built. The default specifies the best volume as defined by the Environment. This parameter is ignored when packages are created.

`Model : String := "!Model.R1000";`

Specifies which links the system will use when it creates a world. The default links are those in !Model.R1000. The null string ("") creates a world with no links.

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

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

Example

The command:

```
library.create ("!users.gzc.my_directory");
```

creates a directory called My_Directory in the world !Users.Gzc.

References

procedure Create_Directory

procedure Create_Unit

procedure Create_World

---

renamed procedure Create_Directory

```
procedure Create_Directory
                  (Name     : Library.Name  := ">>DIRECTORY NAME<<";
                   Kind     : Library.Kind  := Library.Directory;
                   Vol      : Volume        := Library.Nil
                   Model    : String        := "";
                   Response : String        := "<PROFILE>") renames Create;
```

Description

Creates a directory with the specified name in the current library.

Messages and errors are reported in the current log. This procedure renames the Create procedure and has the same parameter profile, with the exception of the Model parameter.

Parameters

`Name : Library.Name := ">>DIRECTORY NAME<<";`

Specifies the name of the directory to create. The parameter placeholder ">>DIRECTORY NAME<<" must be replaced or an error will result.

`Name : Library.Kind := Library.Directory;`

Specifies the type of unit to create, in this case a directory.

`Vol : Volume := Library.Nil;`

Specifies the disk volume on which the object is built. The default specifies the best volume as defined by the Environment.

`Model : String := "";`

Specifies which links the system will use when it creates a world. Since directories do not contain links, none are specified.

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

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

Example

The command:

```
library.create_directory ("!users.gzc.my_directory");
```

creates a directory called My_Directory in the world !Users.Gzc.

---

renamed procedure Create_Unit

```
procedure Create_Unit (Name     : Library.Name := ">>ADA NAME<<";
                       Kind     : Library.Kind := Library.Subpackage;
                       Vol      : Volume       := Library.Nil
                       Model    : String       := "";
                       Response : String       := "<PROFILE>")
                                                    renames Create;
```

Description

Creates a package specification and a body with the indicated name in the current library and installs them.

Messages and errors are reported in the current log. This is a rename of the Create procedure.

Parameters

`Name : Library.Name := ">>ADA NAME<<";`

Specifies the name of the unit to be created. The parameter placeholder ">>ADA NAME<<" must be replaced or an error will result.

`Kind : Library.Kind := Library.Subpackage;`

Specifies the kind of object to be built. The default is a subpackage.

`Vol : Volume := Library.Nil;`

This parameter is ignored when packages are created.

`Model : String := "";`

Specifies which links the system will use when it creates a world. Since subpackages do not contain links, none are specified.

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

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

Example

The command:

```
library.create_unit ("!users.gzc.my_unit");
```

creates a package specification and a body called My_Unit in the world !Users.Gzc and promotes them to the installed state.

---

renamed procedure Create_World

```
procedure Create_World (Name     : Library.Name  := ">>WORLD NAME<<";
                        Kind     : Library.Kind  := Library.World;
                        Vol      : Volume        := Library.Nil
                        Model    : String        := "!Model.R1000";
                        Response : String        := "<PROFILE>")
                                                      renames Create;
```

Description

Creates a world with the specified name on the specified volume.

Errors are reported in the current log. This is a rename of the Create procedure.

Parameters

`Name : Library.Name := ">>WORLD NAME<<";`

Specifies the name of the world to create. The parameter placeholder ">>WORLD NAME<<" must be replaced or an error will result.

`Kind : Library.Kind := Library.World;`

Specifies the kind of unit to be built. The default is a world.

`Vol : Volume := Library.Nil;`

Specifies the volume on which to create the world. The default value specifies that the world should be created on the volume with the most free space.

`Model : String := "!Model.R1000";`

Specifies which links the system will use when it creates the world. The default links are those in !Model.R1000.

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

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

Example

The command:

```
library.create_world ("!users.gzc.my_world");
```

creates a world called My_World in !Users.Gzc. The set of links for this world will be copied from the ones located in !Model.R1000.

---

procedure Default

```
procedure Default (Existing : Name   := "<SELECTION>";
                   Response : String := "<PROFILE>");
```

Description

Sets and displays the name of the default version of the specified object.

This procedure can set any existing version of an object to be the default version of that object. If no new version is specified, or after the new default version is set, the name of the default version is displayed in the current log.

Note that only the current version of directories and worlds exists. No deleted versions of directories or worlds are retained.

Parameters

`Existing : Name := "<SELECTION>";`

Specifies the name of the object for which the default version is to be set or displayed. The default is the current selection.

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

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

Example

A user named Gzc has an Ada unit called Unit_1, which she copied from another directory. The original version ('V(1)) contained several errors, which she repaired in a second version ('V(2)). Since then she has made several modifications that have rendered the unit useless. In the process, she created a third version of the file ('V(3)). To return to the version that was working correctly ('V(2)), she could use the command:

```
library.default ("!users.gzc.unit_1'v(2)");
```

constant Default_Keep_Versions

```
Default_Keep_Versions  : constant := -1;
```

Description

Defines a constant that represents the default number of deleted versions to keep when expunging versions.

When used for setting the retention count for an object, this constant represents the default retention count for the parent object.

The default number of retained versions is 2—the current version and one deleted version.

renamed procedure Delete

```
procedure Delete (Existing : Name                        := "<SELECTION>";
                  Limit    : Compilation.Change_Limit    := "<DIRECTORIES>";
                  Response : String                      := "<PROFILE>");
                                                renames Compilation.Delete;
```

Description

Deletes the default version of the selected or named object.

This procedure deletes the selected or named object. If the deletion of the object would make any other objects obsolete, the deletion is abandoned.

The procedure allows a set of objects to be specified. The combination of wildcards and classes allows the specification of any set of objects in a particular context.

Unrecoverable deletions can be made with the Destroy procedure.

Parameters

`Existing : Name := "<SELECTION>";`

Specifies the name of the object to be deleted. Special names, wildcards, context prefixes, and attributes can be used in this name. The default is the current selection. If no object is selected and the default special name is used, an error will result.

`Limit : Compilation.Change_Limit := "<DIRECTORIES>";`

Specifies which units can be demoted to allow the deletion. The default is to allow demotion only of the units in the same set of directories as the units specified to the operation. See the introduction to this package for the definition of special values such as "<DIRECTORIES>".

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

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

Example

A user named Gzc has in her home directory a file called Test_File that she no longer needs. To remove the file from the directory, she would enter the command:

```
library.delete ("!users.gzc.test_file");
```

This file would be recoverable with the Undelete procedure.

References

renamed procedure Destroy

procedure Undelete

subtype Compilation.Change_Limit

procedure Compilation.Delete

subtype Compilation.Unit_Name

---

renamed procedure Destroy

```
procedure Destroy (Existing  : Name                          := "<SELECTION>";
                   Threshold : Natural                       := 1;
                   Limit     : Compilation.Change_Limit      := "<DIRECTORIES>";
                   Response  : String                        := "<PROFILE>")
                                                  renames Compilation.Destroy;
```

Description

Destroys all of the versions of the selected or named object.

This procedure destroys the selected or named object. If the destruction of the object would make any other objects obsolete, the destruction is abandoned. This procedure is equivalent to deleting and expunging all versions of an object.

The procedure allows a set of objects to be specified. The combination of wildcards and classes allows the specification of any set of objects in a particular context.

Recoverable deletions can be made with the Delete procedure.

Parameters

`Existing : Name := "<SELECTION>";`

Specifies the name of the object to be deleted. Special names, wildcards, context prefixes, and attributes can be used in this name. The default is the current selection.

`Threshold : Natural := 1;`

Specifies the total number of objects per named object that can be destroyed before the procedure fails. The default permits only the named units—not their dependents—to be destroyed.

`Limit : Compilation.Change_Limit := "<DIRECTORIES>";`

Specifies which units can be demoted to allow the deletion. The default is to allow demotion only of the units in the same set of directories as the units specified to the operation. See the introduction to this package for the definition of special values such as "<DIRECTORIES>".

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

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

Example 1

A user named Gzc has in her home directory a text file called Test_File that she no longer needs. To delete and expunge the file from the directory, she would enter the command:

```
library.destroy ("!users.gzc.test_file");
```

The file would not be recoverable with the Undelete procedure.

Example 2

A user name Gzc has an Ada unit called Macros in her home directory. She wants to destroy both the spec and body. To do this, she uses the command:

```
destroy ("macros",2);
```

References

procedure Delete

---

procedure Display

```
procedure Display (Name : Library.Name := "[]");
```

Description

Displays the library containing the named object, with the object selected.

Parameters

```
Name :  Library.Name := "[]";
```

Defines a simple name. The default is the current context.

---

procedure Enclosing_World

```
procedure Enclosing_World (Levels    : Positive := 1;
                           Response  : String   := "<PROFILE>");
```

Description

Changes the context to the parent world of the current context.

Parameters

`Levels : Positive := 1;`

Specifies the number of levels (worlds) to go up in changing the context.

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

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

Example

The current context for a user named Gzc is !Users.Gzc.Sample_World. She wants to change the context to the enclosing world (!Users.Gzc). The command:

```
library.enclosing_world;
```

will accomplish this.

References

EST, procedure Common.Enclosing

renamed exception Error

```
Error : exception renames Profile.Error;
```

Description

Defines the exception raised when an error condition occurs in a command and when the error reaction defined in the profile requests an exception to be raised.

procedure Expunge

```
procedure Expunge (Existing       : Name    := "<IMAGE>";
                   Keep_Versions  : Integer := 0;
                   Recursive      : Boolean := True;
                   Response       : String  := "<PROFILE>");
```

Description

Expunges previously deleted versions of objects, keeping the specified number of versions.

The procedure effectively makes deletions permanent. Deletions can be reversed (undeleted) until the objects are expunged. Once the object is expunged, deleted versions are gone. New versions of Ada units are created when the !Commands-.Common.Edit procedure is executed on the unit.

Some deleted versions of the object can be retained by specifying the value of the Keep_Versions parameter to be some number. In this case, only the specified number of deleted versions is retained (the current version is not counted). The lowest-numbered deleted versions are destroyed and the highest-numbered versions are retained. If a lower-numbered version is the default version, it is retained, even if higher-numbered versions will be deleted.

Parameters

`Existing : Name := "<IMAGE>";`

Specifies the name of the objects to be expunged. If all versions of the object are expunged, the object is destroyed. Special names, wildcards, context prefixes, and attributes are allowed in this name. The default is the current image.

`Keep_Versions : Integer := 0;`

Specifies the number of versions of the object that are not to be expunged. The default is to keep zero deleted versions (to expunge all deleted versions).

`Recursive : Boolean := True;`

Specifies whether to expunge any subobjects of the named object. The default is to expunge all subobjects, as well as the named object.

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

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

Example

A user named Gzc deleted a file called File_1 from her home directory. She can permanently expunge it with the command:

```
library.expunge ("!users.gzc.{file_1}");
```

---

type Field

```
type Field is (Object, Version, Class, Subclass, Updater, Update_Time,
               Creator, Create_Time, Reader, Read_Time, Size, Status,
               Frozen, Retain, Declaration);
```

Description

Defines the set of data that can be displayed for any object.

This type is used to specify what data are displayed about objects in the List procedure and several renames of List. The type is also used to specify what data the listing is sorted by when a display is generated.

Enumerations

`Class`

Specifies to display the class name of the object.

`Create_Time`

Specifies to display the time when the object was created.

`Creator`

Specifies to display the name of the user who created the object.

`Declaration`

Specifies to display the declaration of the object. Sorting by declaration means displaying objects in the order in which they occur in their enclosing package.

`Frozen`

Specifies to display whether the object is frozen.

`Object`

Specifies to display the unique simple name of the object.

`Read_Time`

Specifies to display the time when the object was last read.

Reader

Specifies to display the name of the user who last read the object.

Retain

Specifies to display the number of retained versions of the object.

Size

Specifies to display the size of the object in bytes of data (which includes some overhead).

Status

Specifies to display the declaration state of the object (applies only to Ada class objects).

Subclass

Specifies to display the subclass of the object.

Update_Time

Specifies to display the time that the last update was performed. For Ada objects, this is the time when editing began after the last time the image was committed (saved).

Updater

Specifies to display the name of the user who last updated the object.

Version

Specifies to display the version of the object. An asterisk appears in front of the default version name.

References

procedure Ada_List

procedure File_List

procedure List

procedure Verbose_List

---

type Fields

```
type Fields is array (Field) of Boolean;
```

Description

Defines a type used to specify the set of data to be displayed by the List procedure.

A parameter of this type in the List procedure specifies the fields to display. Several constants of this type provide common sets of fields.

References

constant Ada_Format

procedure Ada_List

constant All_Fields

procedure File_List

procedure List

constant Terse_Format

constant Verbose_Format

procedure Verbose_List

renamed procedure File_List

```
procedure File_List (Pattern     : Name    := "<IMAGE>@'C(FILE)";
                     Displaying  : Fields  := Library.Verbose_Format;
                     Sorted_By   : Field   := Library.Object;
                     Descending  : Boolean := False;
                     Response    : String  := "<PROFILE>";
                     Options     : String  := "") renames List;
```

Description

Displays the specified set of data for the specified set of file objects.

The procedure performs exactly as the List procedure except that this procedure has different default parameters. The default parameters in this procedure provide a detailed display of all versions of all file class objects in the current context.

Parameters

`Pattern : Name := "<IMAGE>@'C(FILE)";`

Defines the set of objects to be listed. Special names, wildcards, context prefixes, and attributes can be used in this name. The default gives the set of all versions of all file class objects in the default context.

`Displaying : Fields := Library.Verbose_Format;`

Specifies the set of data to display about each object. The default is to display the verbose set.

`Sorted_By : Field := Library.Object;`

Specifies the field that should be sorted to order the list. The default is to order alphabetically by the object name.

`Descending : Boolean := False;`

Specifies whether to reverse the ordering. The default is to use the natural ascending order.

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

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

```
Options :  String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

Example

Consider the following world:

```
!Users.Gzc
  File_1
  Library
  My_Directory
  My_Unit
  My_Unit
  Sample_Directory
  Sample_World
  S_1
  S_1_Switches
```

The command:

```
library.file_list;
```

produces the following results, showing information for the two files in that world. Note that the display under "*Continued . . .*" can be obtained by scrolling to the right.

```
------------------------------------------------------------------------------

!USERS.GZC % LIBRARY.FILE_LIST                              STARTED 5:24:12 PM
------------------------------------------------------------------------------


87/01/06 17:24:17 ::: Listing of !USERS.GZC.<IMAGE>@'C(FILE) sorted by object.

   OBJECT     VER CLASS SUBCLASS UPDATER   UPDATE_TIME      SIZE STATUS FRZ
============ === ===== ======== ======= ================== ==== ====== ===
FILE_1       *2  FILE   TEXT       GZC   87/01/06 16:02:15  37   n/a
S_1_SWITCHES *4  FILE   SWITCH     GZC   87/01/06 17:23:23 303   n/a

87/01/06 17:24:19 ::: [End of Library.List command -- No errors detected].

Continued . . .

RETAIN
======
1
1
```

---

procedure Freeze

```
procedure Freeze (Existing  : Name    := "<IMAGE>";
                  Recursive : Boolean := True;
                  Response  : String  := "<PROFILE>");
```

Description

Freezes the specified object to prevent further changes to it.

Freezing prevents changes to an object or set of objects, including changing their state. The procedure can freeze a single object or an entire directory structure of objects.

Freezing objects can be used as part of releasing software. Once an object is frozen, no changes can be made to the object. However, a frozen object can be executed. The Unfreeze procedure undoes the effect of this procedure.

Many library operations affect more than one object (an object and its containing directory or world, or a unit and its dependents). If any of these objects are frozen, the operation fails.

The Freeze and Unfreeze procedures are both subject to access control restrictions. For a user to freeze or unfreeze objects within a world, the user must have owner access to that world.

Parameters

`Existing : Name := "<IMAGE>";`

Specifies the object to be frozen. The default is the current image. Special names, context prefixes, wildcards, and attributes can be used to specify any series of objects to be frozen.

`Recursive : Boolean := True;`

Specifies whether to freeze subobjects. The default is to freeze subobjects. To avoid freezing subworlds, use Recursive => False and the wildcard ?.

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

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

Example

A user has in his home directory a world called Sample_World that he wants to freeze. To accomplish this, he can use the following command:

```
library.freeze ("sample_world");
```

References

procedure List

procedure Unfreeze

type Kind

```
type Kind is (World, Directory, Subpackage);
```

Description

Defines the major structural elements that can be created to produce the directory system.

Enumerations

Directory

Defines a directory as a structural element in the directory system. Directories provide a structural element that is also like an Ada library. Directories and worlds are collectively called libraries. A directory provides a closed naming scope as does a library. It also requires explicit imports and exports through a set of links. A directory, however, uses the set of links provided by the enclosing world; a directory does not have its own links.

Directories are used for structuring or partitioning the contents of a world. Because the world contains all of the links for the entire structure, directories are better suited to breaking down the project or system within the world.

Directories can be nested inside other directories or worlds. Directories can contain any set of units such as directories, worlds, Ada units, files, sessions, or other Environment-defined objects.

Subpackage

Defines a standard Ada package as a structural element in the directory system. A subpackage is an element of its containing library. It resides on the same disk volume as its enclosing world. Subpackages do not have links.

World

Defines a world as a structural element in the directory system. Worlds provide a structural element that is like an Ada library. Directories and worlds are collectively called libraries. A world provides a closed naming scope as does a library. A world requires explicit imports and exports through a set of links. Each world has its own links. These links provide imports and exports not only for the world but also for any contained directory. In addition, these links provide name resolution for units within the world or within contained directories.

Worlds are used as the resource management element. The effect of links and the closed scoping makes a world the focal point for name resolution and resource allocation. The internal links associated with a world allow any unit in any directory within the world to have visibility to other units. Thus the world and its associated set of links can capture the entire set of names. For further information, see the Key Concepts in this book.

Worlds can be nested inside other directories or worlds. Worlds can contain any set of units such as directories, worlds, Ada units, files, sessions, or other Environment-defined objects (see the Key Concepts in this book).

References

procedure Create

---

procedure List

```
procedure List (Pattern     : Name    := "<IMAGE>@";
                Displaying  : Fields  := Library.Terse_Format;
                Sorted_By   : Field   := Library.Object;
                Descending  : Boolean := False;
                Response    : String  := "<PROFILE>";
                Options     : String  := "");
```

Description

Displays the specified set of data about the specified set of versions of specified objects.

This procedure displays a list of objects and data about those objects in the current log. The list can be of any set of objects, sorted in any specified way.

The default parameters produce an alphabetic list of objects with no additional information. Renamed versions of this procedure with different default parameters give other forms of lists.

Parameters

`Pattern : Name := "<IMAGE>@";`

Defines the set of objects to be listed. Wildcards, context prefixes, and attributes can be used in this name. The default gives the set of objects in the default context.

`Displaying : Fields := Library.Terse_Format;`

Specifies the set of data to display about each object. The default is to list only the name of the object.

`Sorted_By : Field := Library.Object;`

Specifies the field that should be sorted to order the list. The default is to order alphabetically by the object name.

`Descending : Boolean := False;`

Specifies whether to reverse the ordering. The default is to use the natural ascending order.

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

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

```
Options :  String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

Example

A user wants to display an alphabetical listing of all files in his home world that end with the string "_1". To accomplish this, he can use the command:

```
library.list ("@_1");
```

---

procedure Move

```
procedure Move (From        : Name    := "<REGION>";
                To          : Name    := "<IMAGE>";
                Recursive   : Boolean := True;
                Response    : String  := "<PROFILE>";
                Copy_Links  : Boolean := True;
                Options     : String  := "");
```

Description

Moves (copies) the value of an existing object to another object and then deletes the existing object.

This procedure can move a single object or a hierarchy of objects. Wildcards can be used to specify a set of objects to be moved. The Recursive parameter allows any subobjects of the named object to be moved. If more than one object is moved, each object is moved independently of any other.

This procedure creates a new version of the existing object if the new object already exists. This may force old versions of the object to be expunged.

If the new object does not exist, it is created. For all Ada units, the new object is created in the source state.

If any move would result in demotion or obsolescence of an existing object, the deletion of that object is abandoned but is still copied. Other objects being moved by this procedure that do not make other units obsolete are not affected. Objects that cannot be copied will not be deleted.

The semantic consistency of the new object is not assured by this procedure. Semantic references must be checked after the object is moved.

This procedure is used for the following purposes:

- To move an object from one directory to another, giving it the same or a different simple name in its new location.
- To copy the links that are associated with each world. The set of links for a world does not have a name, so the set of links is not copied as an object in a world. The Copy_Links parameter allows the procedure to copy those links.

Table 7–3 illustrates the results of executing the Move procedure with various types of objects as the To and From parameters. The To parameter objects are shown horizontally and the From parameter objects are shown vertically in the table.

The word TO indicates that the object specified by the To parameter is moved to the object specified by the From parameter. The word INTO indicates that the

object specified by the From parameter is moved into the object specified by the To parameter.

The number in parentheses following the results indicates a restriction on the move. These restrictions are listed below the table.

Table 7-3. Using the Move Procedure with To and From Parameters

			To Parameter			
From Parameter	**Non-Ada object**	**Library unit**	**Subunit**	**World**	**Directory**	**No object**
Non-Ada object	TO (1)	Error	Error	INTO	INTO	TO
Library unit (2)	Error	TO	TO	INTO	INTO	TO
Subunit (2)	Error	INTO	TO	INTO	INTO	TO
World (3)	Error	Error	Error	TO (4)	TO (4)	TO
Directory (3)	Error	Error	Error	TO (4)	TO (4)	TO

(1) Objects must be of the same class.

(2) If the Recursive is true, the subunits of the unit are involved. The relative nesting of subunits is preserved.

(3) If the Recursive is true, the subcomponents of the unit are involved. The relative nesting of subunits is preserved.

(4) The contents of the From library are merged with contents of the To library.

Parameters

```
From : Name := "<REGION>";
```

Specifies the existing object or objects to be moved. The name can use special names, wildcards, context prefixes, and attributes. The default is the current selection, whether or not the cursor is in the selection.

```
To : Name := "<IMAGE>";
```

Specifies the name of the new object. If the name exists, the old value of the object is deleted. If the name does not exist, it is created.

```
Recursive : Boolean := True;
```

Specifies whether to move subunits. The default is to move all subunits.

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

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

```
Copy_Links : Boolean := True;
```

Specifies whether to copy the links that are associated with any world that is moved. The default is to copy all links.

```
Options : String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

Example

Consider the following world:

```
!User.Rjb
---------
    Check_Messages  : Ada (Proc_Spec);
    Check_Messages  : Ada (Proc_Body);
    Macros          : Library;
```

It is decided that the procedure belongs in the Macros library instead of the home world and should be called Check. To move the procedure and rename it, the command:

```
move ($check_messages","$macros.check");
```

is executed. The procedure is copied into the Macros library and then deleted from the world Rjb. The result is that the world Rjb contains only the Macros library, and the full name of the procedure is now !Users.Rjb.Macros.Check.

References

procedure Copy

---

subtype Name

```
subtype Name is String;
```

Description

Defines a name for objects used in procedures in this package.

The type allows special names, wildcards, context prefixes, attributes, and substitution characters. In some uses of this type, the name must be unique. See the Key Concepts in this book for more information about naming in general.

constant Nil

```
Nil : constant Volume := Volume'First;
```

Description

Defines the constant disk volume that represents the "best" volume.

procedure Reformat_Image

```
procedure Reformat_Image (Existing : Name   := "<SELECTION>";
                          Response : String := "<PROFILE>");
```

Description

Forces the creation of a new pretty-printed image of an Ada unit.

Ada units have two representations: the DIANA tree representation and the pretty-printed image. This procedure forces the creation of a new pretty-printed image of an Ada unit.

Use of this procedure is necessary if pretty-printing switches are changed and the user wants to force a unit that was pretty-printed using the old switch settings to be pretty-printed with the new switch settings.

Parameters

`Existing : Name := "<SELECTION>";`

Specifies the object whose image is to be reformatted. The default is the current image. Special names, context prefixes, wildcards, and attributes can be used to specify any series of objects to be reformatted.

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

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

procedure Rename

```
procedure Rename (From     : Name        := "<SELECTION>";
                  To       : Simple_Name := ">>NEW SIMPLE NAME<<";
                  Response : String      := "<PROFILE>");
```

Description

Renames the specified existing object to the specified new name.

This procedure creates an object of the new name that contains all of the contents of the old name. All contained directories, worlds, links, and objects are moved into the new object.

If the new name already exists, a new version of the object is created.

If the new object does not exist, it is created. For all Ada units, the new object is created in the source state.

Parameters

`From : Name := "<SELECTION>";`

Specifies the existing object or objects to be renamed. Special names, wildcards, context prefixes, and attributes are allowed in this name. The default is the current selection.

`To : Simple_Name := ">>NEW SIMPLE NAME<<";`

Specifies the name of the new, renamed object. Substitution characters are allowed in this name. The default parameter placeholder ">>NEW SIMPLE NAME<<" must be replaced or an error will result.

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

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

Restrictions

If any renaming would result in demotion or obsolescence of an existing object, the object is not deleted but is still copied into the new name. Other objects being renamed by this procedure that do not make other units obsolete are not affected. Objects that cannot be copied are not deleted.

The semantic consistency of the new object is not assured by this procedure. Semantic references must be checked after the object is renamed.

This procedure will not rename across libraries.

Example

Consider the following world:

```
!Users.Rjb
---------
    Ex : Ada (Pack_Spec);
    Ex : Ada (Pack_Body);
```

It was decided that a long name for package Ex would be useful, and the command:

```
rename ("ex","examples");
```

renames the package. The world now looks like this:

```
!Users.Rjb
---------
    Examples : Ada (Pack_Spec);
    Examples : Ada (Pack_Body);
```

procedure Resolve

```
procedure Resolve (Name_Of       : Name    := "<TEXT>";
                   Target_Name   : Name    := "";
                   Objects_Only  : Boolean := True;
                   Response      : String  := "<PROFILE>");
```

Description

Resolves the selected or specified name and displays its full pathname in the current log.

This procedure takes the specified name or the object that is currently selected and displays the full pathname of that object. By default, the procedure searches for objects of that name. If the Objects_Only parameter is false, identifiers within Ada units also can be resolved.

Parameters

`Name_Of : Name := "<TEXT>";`

Specifies the name to be resolved. The default is the currently highlighted text.

`Target_Name : Name := "";`

Specifies a name that, if it contains substitution characters, produces a new name for the resolved name.

`Objects_Only : Boolean := True;`

Specifies whether to restrict the resolution of names to objects or to include Ada declarations, too. If true, the default, only library units or subunits are resolved.

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

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

procedure Set_Retention_Count

```
procedure Set_Retention_Count
                (Existing       : Name    := "<IMAGE>"
                 Keep_Versions  : Integer := Library.Default_Keep_Versions;
                 Recursive      : Boolean := True;
                 Response       : String  := "<PROFILE>");
```

Description

Sets the retention count for the current image or specified object(s).

This procedure sets the retention count, which is the number of deleted versions of an object that are retained when a new version is created. The retention count does not include the single current (undeleted) version of that object.

The default makes the selected or named object's retention count the same as its parent's retention count.

Parameters

`Existing : Name := "<IMAGE>";`

Specifies the name of the object for which the new retention count is desired. The default is the current image.

`Keep_Versions : Integer := Library.Default_Keep_Versions;`

Specifies the new retention count. The default is to keep the parent's retention count.

`Recursive : Boolean := True;`

Specifies whether to change the retention count of contained units. The default is to change the retention count of the contained units.

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

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

Example

A user named Bill wants to keep five versions of objects within his home directory structure. To accomplish this, in a Command window below his home directory, he can execute the command:

```
library.set_retention_count  ("!users.bill",5);
```

procedure Set_Subclass

```
procedure Set_Subclass (Existing     : Name   := "<SELECTION>";
                        To_Subclass  : String := "";
                        Response     : String := "<PROFILE>");
```

Description

Sets the subclass of the selected or named object.

If a null string value ("") is provided for the To_Subclass parameter, the Environment deduces the subclass.

Note that this procedure is typically used to convert from an older release of the Environment that did not support subclasses. If subclasses are not set on such systems, the information on the library object subclass will not be available for objects created under the old release. For information on subclasses, see the Key Concepts in this book.

Parameters

`Existing : Name := "<SELECTION>";`

Specifies the name of the object(s) for which to set the subclass. The default is the current selection. The name can contain any legal directory name and can match multiple objects.

`To_Subclass : String := "";`

Specifies the name of the subclass to set for the object(s). The null string value ("") specifies that the Environment should deduce the subclass for the object.

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

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

subtype Simple_Name

```
subtype Simple_Name is String;
```

Description

Defines a simple name for objects.

This subtype can use wildcards, but it must be unqualified. It cannot use attributes.

procedure Space

```
procedure Space (For_Object    : Name         := "<IMAGE>";
                 Recursive     : Boolean      := True;
                 Each_Object   : Boolean      := False;
                 Each_Version  : Boolean      := False;
                 Space_Types   : Boolean      := True;
                 Response      : String       := "<PROFILE>"
                 Options       : String "");
```

Description

Displays the disk space, in pages, consumed by the specified object(s).

This procedure displays disk utilization information for each specified object. The information is displayed in the current log. Totals are also displayed when appropriate. All figures are in pages (8,192 bits or 1,024 bytes per page).

For worlds, space for the set of links for that world is also included. Nested worlds, directories, and any other objects to be included must be explicitly specified to be included.

Parameters

`For_Object : Name := "<IMAGE>";`

Specifies the name of the object(s) about which to display disk space information. Special names, wildcards, context prefixes, and attributes are allowed in this name. The default is the current image.

`Recursive : Boolean := True;`

Specifies whether to include subobjects. The default is to include subobjects.

`Each_Object : Boolean := False;`

Specifies whether to display the space for each object of the specified set or the summary of the space used. The default is to display the summary.

`Each_Version : Boolean := False;`

Specifies whether to display information about each version of each object. The default is to display information about the default versions only.

```
Space_Types : Boolean := True;
```

Specifies whether to break down the information into the different types of space used. The default is to break down the information.

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

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

```
Options : String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

---

constant Terse_Format

```
Terse_Format : constant Fields := Fields'(Object => True, others => False);
```

Description

Defines a constant that specifies that a terse set of data be displayed by the List procedure.

procedure Undelete

```
procedure Undelete (Existing : Name   := "<CURSOR>";
                    Response : String := "<PROFILE>");
```

Description

Undoes the deletion (done with the Delete procedure), if any, of the specified version of the specified object.

This procedure deletes the current version of the object and undeletes the specified version of that object. This makes the specified version the current version.

If no version is specified and a current version already exists or if the version specified is the current version, the procedure has no effect. If no version is specified and no current version exists (the object is deleted), the procedure undeletes the highest-numbered (MAX) version.

Parameters

`Existing : Name := "<CURSOR>";`

Specifies the version of the object to be undeleted. Special names, wildcards, context prefixes, and attributes are allowed in this name. The default is the object on which the cursor is located.

The name should include the version attribute to specify which version is to be undeleted. See the Key Concepts in this book for attributes.

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

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

Example

A user accidentally deletes a file called File_1. To undelete it, the user enters the command:

```
library.undelete ("file_1)";
```

The brackets indicate that the file is deleted but not expunged.

References

procedure Delete

procedure Unfreeze

```
procedure Unfreeze (Existing  : Name    := "<IMAGE>";
                    Recursive : Boolean := True;
                    Response  : String  := "<PROFILE>");
```

Description

Unfreezes the named or current object to allow further changes to it.

This procedure undoes the effect of the Freeze procedure. If an object is frozen, no changes can be made to the object. This procedure unfreezes the object so that changes can be made to it.

Freezing objects can be used as part of releasing software. Once an object is frozen, no changes can be made to the object. However, a frozen object can be executed. The Freeze procedure is the opposite of this procedure.

Many operations may change more than one object (an object and its containing directory or world, or a unit and its dependents). If any of these objects are frozen, the operation fails.

The List procedure can be used to determine whether an object is frozen.

The Freeze and Unfreeze procedures are subject to access control restrictions. To freeze or unfreeze objects within a world, a user must have owner access to that world.

Parameters

`Existing : Name := "<IMAGE>";`

Specifies the object to be unfrozen. The default is the current image. Special names, context prefixes, wildcards, and attributes can be used to specify any series of objects to be unfrozen.

`Recursive : Boolean := True;`

Specifies whether to unfreeze subunits. The default is to unfreeze subunits. To avoid unfreezing nested worlds, use Recursive => false and the wildcard ?.

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

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

Example

A user has frozen world called Sample_World, which she now wants to unfreeze. To do this, she can execute the command:

```
library.unfreeze ("sample_world");
```

References

procedure Freeze

constant Verbose_Format

```
Verbose_Format : constant Fields := Fields'(Object .. Update_Time => True,
                                            Size .. Retain        => True,
                                            others                => False);
```

Description

Defines a constant that specifies a verbose set of data to be displayed by the List procedure.

renamed procedure Verbose_List

```
procedure Verbose_List (Pattern     : Name    := "<IMAGE>{@'V(ALL)}";
                        Displaying  : Fields  := Library.Verbose_Format;
                        Sorted_By   : Field   := Library.Object;
                        Descending  : Boolean := False;
                        Response    : String  := "<PROFILE>";
                        Options     : String  := "") renames List;
```

Description

Displays the specified set of data about the specified set of versions of specified objects.

This procedure is similar to the List procedure, but it has different default parameters. The default parameters in this procedure provide a detailed display of all objects in the current context, sorted by object name.

Further explanation and examples can be found in the Key Concepts in this book.

Parameters

`Pattern : Name := "<IMAGE>{@'V(ALL)}";`

Defines the set of objects to be listed. Special names, wildcards, context prefixes, and attributes can be used in this name. The default gives the set of objects, all versions of both deleted and undeleted objects, in the current image.

`Displaying : Fields := Library.Verbose_Format;`

Specifies the set of data to display about each object. The default is to display the verbose set.

`Sorted_By : Field := Library.Object;`

Specifies the field that should be sorted to order the list. The default is to order alphabetically by the object name.

`Descending : Boolean := False;`

Specifies whether to reverse the ordering. The default is to use the natural ascending order.

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

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

```
Options :  String := "";
```

No options currently are implemented for this command. This parameter is reserved for future development.

Example

The command:

```
library.verbose_list ("!users.gzc.@");
```

produces the following results. Note that the display under "*Continued . . .*" can be obtained by scrolling to the right.

```
------------------------------------------------------------------------------

!USERS.GZC % LIBRARY.VERBOSE_LIST  STARTED 6:11:38 PM
------------------------------------------------------------------------------


87/01/06 18:11:40 ::: Listing of !USERS.GZC.@ sorted by object.

     OBJECT        VER  CLASS   SUBCLASS   UPDATER     UPDATE_TIME     SIZE
================== === ======= ========== ======= ================== ====
FILE_1             *2  FILE    TEXT         GZC    87/01/06 16:02:15   37
LIBRARY            *1  LIBRARY DIRECTORY    GZC    87/01/06 14:57:46 8660
MY_DIRECTORY       *1  LIBRARY DIRECTORY    GZC    87/01/06 15:50:37 7302
MY_UNIT            *1  ADA     PACK_SPEC    GZC    87/01/05 14:38:43 7162
MY_UNIT'BODY       *1  ADA     PACK_BODY    GZC    87/01/05 14:38:44 7195
SAMPLE_DIRECTORY   *1  LIBRARY DIRECTORY    GZC    87/01/06 14:57:47 7535
SAMPLE_WORLD       *1  LIBRARY WORLD        GZC    87/01/06 14:57:47 7302
S_1                *1  SESSION NIL                  87/01/06 17:23:18    0
S_1_SWITCHES       *4  FILE    SWITCH       GZC    87/01/06 17:23:23  303
UNIT_1             *1  ADA     PACK_SPEC    GZC    87/01/06 15:52:40 7162
UNIT_1'BODY        *1  ADA     PACK_BODY    GZC    87/01/06 15:52:40 7195

87/01/06 18:11:43 ::: [End of Library.List command -- No errors detected].
```

Continued . . .

```
STATUS    FRZ  RETAIN
=======   ===  ======
n/a            5
n/a            1
n/a            1
INSTALLED      5
INSTALLED      5
n/a            1
n/a            1
n/a            5
n/a            5
SOURCE         5
SOURCE         5
```

---

subtype Volume

```
subtype Volume is Natural range 0 .. 31;
```

Description

Defines the subtype used to represent disk volumes.

This subtype is used to specify the disk units on which libraries are built. The directory or world and all of its subunits reside on a single disk. Whenever a directory or world (both are libraries) is built, it is assigned to a disk volume. This subtype defines the possible disk volumes.

Typical systems contain either two or four disk volumes, numbered consecutively from 1. Volume 0 is not an actual disk; it is used as a convention to indicate that the system should select the volume on which to allocate the control point.

end Library;

package Links

Links are the mechanism provided by the Environment that permit visibility between the Ada units within a world and the units outside that world. They also provide visibility between the units that reside within a world. Links are associated with worlds only and are accessible from all Ada units in a given world.

A link is a mapping between an Ada simple name and the full pathname of an Ada unit in the library system. A possibly empty set of links is associated with each world in the Environment library structure. Each library in the Environment is associated with the set of links of its closest enclosing world. That is, a world is associated with its own set of links, and a directory is associated with the set of links of the closest enclosing world.

A link to a unit outside a world is called an *external link*, and a link to a unit within a world is called an *internal link*. Internal links are created automatically by the Environment unless they are explicitly inhibited by the user (see package Switches); external links are specified by the user.

This package provides operations for creating, changing, deleting, and expunging the set of links associated with a world. This package contains commands for changing links both interactively and programmatically. See the Key Concepts in this book for an introduction to links.

Each link has two parts: a link name and a source name. The link name is a simple Ada identifier by which the link is known in the world. The source name is the pathname of the Ada unit that is associated with the link name. When an Ada unit is with a unit using a link name, the unit actually referenced is the one designated by the source name associated with the link name. Each link name defined for a world must be unique in that world.

The parameters for many of the commands in this package utilize three subtypes: Source_Name, Link_Name, and Source_Pattern. The Source_Name subtype refers to the fully qualified pathname of an Ada unit. For example, the source name for a link to the Environment package Access_List is !Commands.Access_List. (See the Key Concepts in this book for further information on names.) The Link_Name subtype for that package is the name to be used in the *with* clause in an Ada unit within the world containing the link. The default is the Ada simple name of the unit. The user may change that name if desired—for example, when a name has already

been used. Finally, the Source_Pattern subtype is used when the user wants to match a pattern that corresponds to a portion of Source_Name. Pattern-matching characters can be specified for this name. For example, to perform an operation on all units in package !Commands, the user could specify a Source_Pattern of "!Commands?". See the entry for the Source_Pattern subtype for further information on pattern matching.

Commands from Package Common

Many of the operations in package !Commands.Common apply to the set of links associated with a world. Links can be brought into a window with the Links.Edit procedure and then edited with common editing operations that apply to links. The following procedures from package Common apply to links. Other operations from package Common that do not apply to links produce a message to that effect in the Message window when used on links.

These common editing operations are discussed in Editing Specific Types (EST).

procedure Common.Abandon

Ends the editing of the current set of links. The window is removed from the screen. Since all changes to links are made immediately, this procedure does not abandon any of those changes.

procedure Common.Commit

Has no effect, because all changes to links are made immediately. All other operations on links implicitly commit any changes.

procedure Common.Create_Command

Creates a Command window below the current window. If the Command window is created below a window created by the Links.Edit or the Links.Visit command, the *use* clause in the Command window includes package Links. Thus, operations in this package are visible in the Command window without qualification.

procedure Common.Definition

Finds the definition of the selected link or the link on which the cursor is located. This procedure creates or visits a window that contains the specification of the source unit of the selected link.

procedure Common.Edit

Creates a Command window and places in it the command:

```
Update ("selected or current link");
```

where *selected or current link* is the link on which the cursor is currently located, whether or not there is a selection. Providing a new parameter and promoting the command changes the source name for that link.

procedure Common.Elide

Selects which type of link is displayed in the window. This procedure cycles the display from all links (the default) to external links and then to internal links.

procedure Common.Enclosing

Finds the world that contains the links that are in the current window. This procedure creates a window that contains the listing of that world.

procedure Common.Expand

Selects which type of link is displayed in the window. This procedure cycles the display from internal links to external links and then to all links (the default).

procedure Common.Explain

Inserts an explanation below the current link that explains what units use the linked unit. This procedure is useful for determining what dependencies on links exist. If there already is an explanation explaining the link, this procedure removes that explanation.

procedure Common.Release

Ends the editing of the current set of links. The window is removed from the screen.

procedure Common.Revert

Redraws the set of links in the current window. If the set of links has been changed by another user or program, the new image reflects those changes.

procedure Common.Sort_Image

Selects the order in which to display the set of links. This procedure cycles the display from alphabetic by link names (the default), to alphabetic by source names, to alphabetic internal link names followed by alphabetic external link names, and to alphabetic internal source names followed by alphabetic external source names.

procedure Common.Object.Child

If no link is selected, the procedure selects the link on which the cursor is located. If a single link is already selected, the procedure has no effect. If all links are already selected, the procedure selects the link on which the cursor is located.

procedure Common.Object.Copy

Copies a selected link from one set of links to the set of links on which the cursor is located. If the selected link and the cursor are both in the same set of links, the procedure has no effect.

procedure Common.Object.Delete

Deletes the selected link.

procedure Common.Object.First_Child

Selects the first link in the set of links.

procedure Common.Object.Insert

Creates a Command window and places in it the command:

```
Insert ("[link=>] source; etc.");
```

where the link parameter must be specified to provide a new link. Specifying a source for a new link and promoting the command inserts a new link with the same simple name as the source unit. Multiple links can be inserted with one command by separating them with semicolons.

procedure Common.Object.Last_Child

Selects the last link of the set.

procedure Common.Object.Move

Copies a selected link from one set of links to the set of links on which the cursor is located. Currently, the procedure copies the link but does not move it. If the selected link and the cursor are both in the same set of links, the procedure has no effect.

procedure Common.Object.Next

Selects the next link. If no link is already selected, the procedure selects the link on which the cursor is located. If all links are selected, this procedure produces an error.

procedure Common.Object.Parent

Selects the link on which the cursor is located. If no link is already selected, the procedure selects the link on which the cursor is located. If a link is selected, the procedure selects all links in the set. Otherwise, the procedure has no effect.

procedure Common.Object.Previous

Selects the previous link. If no link is already selected, the procedure selects the link on which the cursor is located. If all links are selected, this procedure produces an error.

procedure Add

```
procedure Add (Source   : Source_Name  := ">>SOURCE NAMES<<";
               Link     : Link_Name    := "#";
               World    : World_Name   := "<!MAGE>"
               Response : String       := "<PROFILE>");
```

Description

Adds a link or links to the set of links of the specified world.

The procedure adds a link with the specified link name and source to the specified world. If a link is added and the link name already exists, the source names must reference the same unit.

The same operation is performed by the Insert procedure, except that it brings up a window containing the set of links if one does not already appear on the screen.

Parameters

`Source : Source_Name := ">>SOURCE NAMES<<";`

Specifies the source of the linked unit. Wildcards can be used in this name to specify multiple source units. The default parameter placeholder `">>SOURCE NAMES<<"` must be replaced or an error will result.

`Link : Link_Name := "#";`

Specifies the name of the link. Substitution characters can be used in this name to allow building link names from source names. The default is the simple name of the source unit.

`World : World_Name := "<!MAGE>";`

Specifies the world to which to add the link. The default is the current image.

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

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

Errors

If a set of links is being added and any one fails because of name conflicts with existing units, only that link addition will fail.

Example

The command:

```
links.add ("!commands.@","#","!users.bob");
```

adds links in the world !Users.Bob to the packages in !Commands. The name of each link will be the simple name of the source unit.

References

procedure Insert

procedure Replace

procedure Update

SJM, package Profile

---

constant Any

```
Any : constant Link_Kind := Links_Implementation.Any;
```

Description

A constant specifying that internal or external links, or both, are allowed.

procedure Copy

```
procedure Copy (Source_World  : World_Name      := ">>WORLD NAME<<";
                Target_World  : World_Name      := "<!MAGE>";
                Link          : Link_Name       := "@";
                Source        : Source_Pattern  := "?";
                Kind          : Link_Kind       := Links.Any;
                Response      : String          := "<PROFILE>");
```

Description

Copies the links (or some subset of them) from one world into another world.

This procedure adds links to a target world by copying them from a source world. By default, all links from the source world are chosen. Any subset of those links can be specified with wildcards or specific link names. Also, the type of links to be copied can be specified: internal, external, or both.

The links to be copied must match both the link name and the source pattern specified in the procedure. Typically, either the set of link names or the set of source names is restricted to some subset of the links in the set.

Parameters

`Source_World : World_Name := ">>WORLD NAME<<";`

Specifies the world from which the links are to be copied. The default parameter placeholder ">>WORLD NAME<<" must be replaced or an error will result.

`Target_World : World_Name := "<!MAGE>";`

Specifies the world into which the links are to be copied. The default is the current image.

`Link : Link_Name := "@";`

Specifies which links are to be copied. Pattern matching is allowed. The default pattern matches all link names. See the entry for the Source_Pattern subtype for further information on pattern matching.

`Source : Source_Pattern := "?";`

Specifies the source pattern for the links to be copied. The default pattern matches all source names. See the entry for the Source_Pattern subtype for further information on pattern matching.

```
Kind : Link_Kind := Links.Any;
```

Specifies the kinds of links to copied. The default is both internal and external links.

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

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

Restrictions

Context prefixes cannot be used when specifying the source name for the Source parameter.

Example

The command:

```
links.copy ("!users.bob","!users.sam","@","!tools?");
```

copies any links in !Users.Bob that have a source name beginning with !Tools to the set of links in !Users.Sam.

References

SJM, package Profile

---

procedure Delete

```
procedure Delete (Link     : Link_Name       := ">>LINK NAMES<<";
                  Source   : Source_Pattern  := "?";
                  Kind     : Link_Kind       := Links.Any;
                  World    : World_Name      := "<IMAGE>";
                  Response : String          := "<PROFILE>");
```

Description

Deletes the link(s) from a world.

The procedure deletes links that have the specified link name, source pattern, and kind from the specified world. Any set of links can be specified with wildcards or source patterns. Also, the type of links to be deleted can be specified: internal, external, or both.

The links to be deleted must match both the link name and the source pattern specified in the procedure. Typically, either the set of link names or the set of source names is restricted to some subset of the links in the set.

Parameters

`Link : Link_Name := ">>LINK NAMES<<";`

Specifies the link or links to be deleted. Pattern matching is allowed in this name. The default parameter placeholder ">>LINK NAMES<<" must be replaced or an error will result. See the entry for the Source_Pattern subtype for information on pattern matching.

`Source : Source_Pattern := "?";`

Specifies the source pattern of those links to be deleted. Pattern matching is allowed in this name. The default is any source name. See the entry for the Source_Pattern subtype for further information on pattern matching.

`Kind : Link_Kind := Links.Any;`

Specifies the kinds of links to be deleted. The default is both internal and external links.

`World : World_Name := "<IMAGE>";`

Specifies the world whose links are to be deleted. The default is the current image.

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

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

Restrictions

Context prefixes cannot be used when specifying the source name for the Source parameter.

Example

The command:

```
links.delete ("@","!users.bob?",links.any,"!users.sam");
```

deletes links in the world !Users.Sam that link to any units in the world !Users.Bob.

References

constant Any

SJM, package Profile

---

procedure Dependents

```
procedure Dependents (Link     : Link_Name       := "@";
                      Source   : Source_Pattern  := "?";
                      Kind     : Link_Kind       := Links.Any;
                      World    : World_Name      := "$$";
                      Response : String          := "<PROFILE>");
```

Description

Displays the set of Ada units that depend on a specified link.

The procedure displays in the current output window the set of units in the specified world that depend on the specified link name and source name. The dependent units must be in the installed or coded state. Any set of links can be specified using wildcards in link and source names. Also, the type of links to be checked can be specified: internal, external, or both.

The links to be checked must match the link name, the source pattern, and the kind specified in the procedure. Typically, either the set of link names or the set of source names is restricted to some subset of the links in the set.

Parameters

`Link : Link_Name := "@";`

Specifies the link or links to be checked. Pattern matching is allowed in this name. The default pattern matches all link names. See the entry for the Link_Name subtype for further information on pattern matching.

`Source : Source_Pattern := "?";`

Specifies the source name of those links to be checked. Pattern matching is allowed in this name. The default pattern matches all source names. See the entry for the Source_Pattern subtype for further information on pattern matching.

`Kind : Link_Kind := Links.Any;`

Specifies the kinds of links for which to check dependents. The default is both internal and external links.

`World : World_Name := "$$";`

Specifies the world whose links are to be checked. The default is the enclosing world of the current context.

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

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

Restrictions

Context prefixes cannot be used when specifying the source name for the Source parameter.

References

SJM, package Profile

---

procedure Display

```
procedure Display (World    : World_Name      := "<IMAGE>";
                   Link     : Link_Name       := "@";
                   Source   : Source_Pattern  := "?";
                   Kind     : Link_Kind       := Links.Any;
                   Response : String          := "<PROFILE>");
```

Description

Displays the specified set of links for the selected or specified world in the log file.

The procedure displays the links that match the specified link name, the specified source name, and kind for the selected or specified world. The results are placed in Current_Output, which is, by default, the current output window. Also, the type of links to be displayed can be specified: internal, external, or both.

The links to be displayed must match the link name, the source pattern, and link kind specified in the procedure. Typically, either the set of link names or the set of source names is restricted to some subset of the links in the set.

Parameters

`World : World_Name := "<IMAGE>";`

Specifies the world whose links are to be displayed. The default is the current image. If the current image is not a world, the links for the enclosing world are displayed.

`Link : Link_Name := "@";`

Specifies the link name of the links to be displayed. The default pattern matches all links. See the entry for the Link_Name subtype for further information on pattern matching.

`Source : Source_Pattern := "?";`

Specifies the source name of the links to be displayed. The default pattern matches all source names. See the entry for the Source_Pattern subtype for further information on pattern matching.

`Kind : Link_Kind := Links.Any;`

Specifies the kind of links to be displayed. The default is both internal and external links.

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

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

Restrictions

Context prefixes cannot be used when specifying the source name for the Source parameter.

Example

The command:

```
links.display ("!users.bob","@","!tools?");
```

shows all links to !Tools in the set of links for the world !Users.Bob.

References

SJM, package Profile

---

procedure Edit

```
procedure Edit (World : World_Name := "<IMAGE>");
```

Description

Edits the links from the specified world.

The procedure creates a window and displays the set of links from the specified world in that window. If a window already exists with those links in it, the window is reused. From the window, the links can be edited with many operations from package !Commands.Common that apply to this class of object (see the introduction to package Links for details).

This procedure creates a new window for each set of links to be edited. To reuse the same window but to edit a different set of links in that window, see the Visit procedure.

Parameters

```
World :  World_Name := "<IMAGE>";
```

Specifies the world whose links are to be edited. The default is the current image.

Example

The command:

```
links.edit ("!users.jim");
```

displays the set of links for the world !Users.Jim in a window.

References

procedure Visit

---

procedure Expunge

```
procedure Expunge (World    : World_Name := "<IMAGE>";
                   Response : String     := "<PROFILE>");
```

Description

Removes obsolete links from the set of links for the specified world.

When a unit is deleted from a world, the links to that unit are not removed from all of the sets of links that reference the unit, in case the user later wants to restore or undelete the unit. The unit thus can be restored or undeleted without again adding all of the links that reference the unit. When removal of an obsolete link is desired, references to links to units that no longer exist in the specified world can be removed with the Expunge procedure.

Parameters

`World : World_Name := "<IMAGE>";`

Specifies the name of the world whose set of links is to be expunged. The default is the current image. If the current image is not a world, the links for the enclosing world are expunged.

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

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

References

SJM, package Profile

constant External

```
External : constant Link_Kind : Links_Implementation.External;
```

Description

Defines a constant that specifies an external link.

An external link is a link whose source unit is contained outside the current world.

procedure Insert

```
procedure Insert (Source : Source_Name := ">>SOURCE NAME<<");
```

Description

Inserts a link to the units specified by the source name in the set of links.

The procedure creates a link in the current set of links with the specified source name. By default, the link name is created with the same simple name as the source unit. The window that contains the affected set of links is updated.

If a set of links is not being edited when this procedure is executed, the procedure creates a window with the links of the enclosing world (from wherever the current context is) and inserts the new link in that set of links.

The same operation is performed by the Add procedure, without the use of a window.

The parameter placeholder ">>SOURCE NAME<<" must be replaced with a link specification that follows, generally, the format of Options parameters (that is, `Source=> Value`, where the value delimiter can be =>, :=, or =). Multiple link specifications can be given by separating them with a semicolon or a comma. For further information on the syntax of the Options parameter, see the Key Concepts in this book.

Parameters

`Source : Source_Name := ">>SOURCE NAME<<";`

Specifies the source unit of the link. More than one source unit can be specified within this parameter. Multiple names can be separated by a semicolon, and wildcards can be used to denote several units. The default parameter placeholder ">>SOURCE NAME<<" must be replaced or an error will result.

Restrictions

Context prefixes cannot be used when specifying the source name for the Source parameter.

Example

A user named Bill is currently working in his home world. He wants to add a link to a unit called Unit_1 in the home world of a user named Sue. To do this, he executes the command:

```
links.insert ("!users.sue.unit_1");
```

References

procedure Add

procedure Replace

procedure Update

constant Internal

```
Internal : constant Link_Kind : Links_Implementation.Internal;
```

Description

Defines a constant that specifies an internal link.

An internal link is a link that designates a unit contained inside the current world but is not nested inside any contained worlds.

subtype Link_Kind

```
subtype Link_Kind is Links_Implementation.Link_Kind;
```

Description

Specifies the kind of link to delete, replace, or display.

The kind of link can be internal, external, or any (which specifies both).

References

constant Any

constant External

constant Internal

---

subtype Link_Name

```
subtype Link_Name is String;
```

Description

Defines the form of link names.

The subtype is a string that denotes the local names of units (the link names used when *with*ing the unit). The link name is a simple name that must be a legal Ada identifier. Link names are the names of Ada units that are used in context clauses of other Ada units.

When used as a parameter, the name allows use of pattern matching except when used in the Add and Replace procedures. In those two procedures, the link name can contain only substitution characters. In all cases, the link name must resolve to a legal Ada simple name. For further information on naming, see the Key Concepts in this book.

Patterns include the following characters:

- # The pound sign is replaced by a single character. It will not match the null string or a period (.). For example, File_# matches File_1 and File_A.
- @ The *at* sign matches any string that does not contain a period. For example, !Users.Mary.@ matches everything in Mary's home world, but nothing below that level in the library structure. The *at* sign matches the null string.
- ? The question mark matches any sequence of characters at the beginning of the name (that is, ? or !?) or a sequence of characters beginning with a period (.). It matches the null string. For example, !Users? matches everything in !Users.
- [] Brackets indicate a set of objects—for example, [!Users.Mary?,!Users.John?].
- ~ The tilde indicates that something should not be matched—for example, ~!Users.Mary means that !Users.Mary is not matched.

procedure Replace

```
procedure Replace (Source   : Source_Name  := ">>SOURCE NAMES<<";
                   Link     : Link_Name    := "#";
                   World    : World_Name   := "<IMAGE>";
                   Response : String       := "<PROFILE>");
```

Description

Replaces links to the units named by the Source name, if they exist; if they do not exist, they are created.

The same operation is performed by the Update procedure except that the Update procedure brings up a window displaying the set of links for the world. A similar operation is performed by the Add and Insert procedures, except that these require that no link of the same name exist at the time of the operation.

Parameters

`Source : Source_Name := ">>SOURCE NAMES<<";`
Specifies the source of the linked unit. Wildcards can be used in this name for specifying multiple links. The default parameter placeholder `">>SOURCE NAMES<<"` must be replaced or an error will result.

`Link : Link_Name := "#";`
Specifies the new name of the link. Substitution characters can be used in this name to allow building names from source names. The default causes the simple name of the source unit to be the link name.

`World : World_Name := "<IMAGE>";`
Specifies the world in which to replace the link. The default is the current image.

`Response : String := "<PROFILE>";`
Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example

A user named John has in his home world a link named Message to a package called !Users.Sue.Message. He wants to use the link name Message for the Environment package !Commands.Message, whose current link name is Environment_Message. To update the link name, John uses the command:

```
links.replace ("!commands.message","#","!users.john");
```

After executing this command, he may want to delete the link to the same package using the link name Environment_Message. This replace will succeed only if no installed unit in John's world has *with*ed Message.

References

procedure Add

procedure Insert

procedure Update

SJM, package Profile

---

subtype Source_Name

```
subtype Source_Name is String;
```

Description

Defines the form of the names for units that are to become the sources of links.

This subtype is a string that names one or more library units. The units do not have to be installed, but their declaration must exist in the library. All forms of name expressions (context, prefixes, wildcards, attributes, and so on) can be used in Source_Name strings as long as the only objects named are Ada units. In the Update procedure, only one Ada unit can be named.

subtype Source_Pattern

```
subtype Source_Pattern is String;
```

Description

Defines a subtype that contains a complete directory pathname beginning with !.

Wildcards can appear in the pathname. A source pattern string is matched against the source names of the links of a world to select a set of links on which to operate.

Parameters of this type default to ?, which is a wildcard pattern that matches all source names.

Patterns include the following characters:

- # The pound sign is replaced by a single character. It will not match the null string or a period (.). For example, File_# matches File_1 and File_A.
- @ The *at* sign matches any string that does not contain a period. For example, !Users.Mary.@ matches everything in Mary's home world, but nothing below that level in the library structure. The *at* sign matches the null string.
- ? The question mark matches any sequence of characters at the beginning of the name (that is, ? of !?) or a sequence of characters beginning with a period (.). It matches the null string. For example, !Users? matches everything in !Users.
- [] Brackets indicate a set of objects—for example, [!Users.Mary?,!Users.John?].
- ~ The tilde indicates that something should not be matched—for example, ~!Users.Mary means that !Users.Mary is not matched.

procedure Update

```
procedure Update (Source : Source_Name := ">>SOURCE NAME<<");
```

Description

Changes the source name of the selected link to be the one specified.

This command, generated in response to the !Commands.Common.Edit procedure on a links window, usually is not typed directly by the user.

The procedure updates the source name of the selected link in the associated window of links. The window that contains the affected set of links is updated. If a set of links is not being edited when this procedure is executed, an error results.

A similar operation is performed by the Replace procedure, except that it does not require that a window containing the set of links be displayed on the screen.

Parameters

```
Source : Source_Name := ">>SOURCE NAME<<";
```

Specifies the fully qualified source name. Wildcards can be used if they resolve to a single unit. The default parameter placeholder ">>SOURCE NAME<<" must be replaced or an error will result.

Errors

An error will result if a window containing a selected link is not displayed (that is, the command works only in a Command window below the set of links).

Example

A user named John has a link in his home world to the Environment package !Commands.Message. A coworker named Sue has a package in her home world called Message, which John wants to reference by the link name Message. He no longer wants to have a link to the package provided by the Environment. To update the source name for the link name Message with that of the package in Sue's world, John uses the Edit procedure to open a window containing the links for his home world. He then selects the Message link in the display and presses Edit, which produces a prompt for `Links.Update (">>SOURCE NAME<<");`. He replaces `>>SOURCE NAME<<` with `!Users.Sue.Message` and executes the command.

References

procedure Add

procedure Insert

procedure Replace

procedure Visit

```
procedure Visit (World : World_Name := "<IMAGE>");
```

Description

Visits the links from the specified world.

This procedure replaces an existing window containing the links editor with the set of links from the specified world. If a window does not already exist for editing links, one is created. This differs from the Edit procedure, which creates a new window for the links editor. From the window, the links can be edited with many operations from package !Commands.Common that apply to the editing links (see the introduction to package Links for details).

This procedure does not create a new window for each set of links to be edited. To create a new window to edit a different set of links in that window, use the Edit procedure.

Parameters

```
World : World_Name := "<IMAGE>";
```

Specifies the world whose links are to be edited. If there is no selection and the default value is used, the current containing world is used.

References

procedure Edit

---

subtype World_Name

```
subtype World_Name is String;
```

Description

Defines the form of world names.

This subtype is a string that denotes the full name of an existing world. When used as a parameter, the name allows the use of special names, wildcards, and context prefixes. The name can resolve to only one world. For further information on naming, see the Key Concepts in this book.

end Links;

RATIONAL

package Switches

Package Switches provides operations for creating, manipulating, changing, and deleting the sets of switches associated with a library or a session. This package contains commands for changing switches both interactively and programmatically.

Error Response

The commands in this package have a Response parameter that specifies how the command should respond to errors, how to generate logs, and what activities to use. The response profile ("<PROFILE>"), which many commands use by default, specifies the job response profile. If there is no job response profile, the session response profile ("<SESSION_PROFILE>") is used. If there is no session response profile, the system's default profile ("<DEFAULT>") is used. For further information on profiles, see SJM, package Profile.

Special Names

Many of the commands in this package have *special names* as default values to parameters requiring names. Anywhere that a string name can be used, a special name can be used. Special names allow you to designate without supplying a pathname. They take the form "*<special name>*", where *special name* specifies the text, region, or area that they represent, as described below:

"<SELECTION>"	References the highlighted object, if the cursor is located in a highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the object on which the cursor is located, whether or not there is a highlighted area in the window.
"<IMAGE>"	References the highlighted object, if the cursor is in a highlighted area. If the cursor is not located in the highlighted area, this special name references the image on which the cursor is located.
"<TEXT>"	References the object named in the highlighted text in the image in the window.

"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, this special name references that activity rather than the default activity.

You can replace special names with other types of naming expressions, as accepted by that parameter.

Parameter Placeholders

Some of the commands in this package have a parameter placeholder as a default value. Parameter placeholders take the form "*>>name<<*", where *name* is the type of object that should replace the parameter placeholder. For further information, see the Key Concepts in this book.

Overview of Switches

There are two kinds of switches: library and session. ***Library switches*** are defined for a specific library (that is, a directory or world). A set of these switches is associated with a particular library. These switches affect how compilation is done, how links are managed, or how pretty-printing is done in that library. Changes to any of these switches take effect immediately after the change is made. Library switches are documented in this section of the *Rational Environment Reference Manual.*

Session switches affect the way the system behaves on your terminal. Some of these switches are read by the Environment only when the user logs in. Others are effective immediately. Still others are checked when some window is created. Session switches are documented in SJM, Session Switches.

Switches are stored in File class objects with subclass Switch. Each switch file contains all the switches for a given session or library. A session switch file is located in the user's home directory. You can have several switch files, one for each session. Each library may have its own library switch file, or several libraries may share the same switch file.

A session switch file can be edited using the Edit_Session_Attributes procedure. A library switch file can be edited using the Edit procedure. If there is no library switch file associated with that library, the editor will create one.

After either the Edit_Session_Attributes or the Edit command is entered, the Environment will open a window showing the list of switches. To change a switch, place the cursor on the line displaying the switch and select it using [Object] - [←]. Press [Edit]. If the switch has a Boolean value, the value will be toggled. If the switch takes a non-Boolean value, a Command window will be opened with prompts to insert the new value.

To get help on a switch, enter the switch file as before, place the cursor on the line of the switch in question, and press [Explain]. A help file will appear on the line below the switch.

The following list names all library switches, grouped by functionality. Following the list are descriptions of the switches, ordered alphabetically.

Library Switches Grouped by Function

Switches for Ada Units

Alignment_Threshold
Comment_Column
Consistent_Breaking
Create_Subprogram_Specs
Enable_Deallocation
Id_Case
Ignore_Interface_Pragmas
Ignore_Minor_Errors
Ignore_Unsupported_Rep_Specs
Keyword_Case
Line_Length
Major_Indentation
Minor_Indentation
Number_Case
Page_Limit
Statement_Indentation
Statement_Length
Wrap_Indentation

Switches for Networking

Account
Auto_Login
Password
Prompt_For_Account
Prompt_For_Password
Remote_Directory
Remote_Machine
Remote_Roof
Remote_Type
Send_Port_Enabled
Transfer_Mode
Transfer_Structure
Transfer_Type
Username

Switches for Links

Create_Internal_Links
Require_Internal_Links

Switches for Listings

Asm_Listing
Seg_Listing
Terminal_Echo

Library Switch Descriptions

Account

Specifies an account name to set up an FTP connection between a local and a remote computer. The account is on the remote computer. The default is the null string. The full switch name is Ftp.Account.

Alignment_Threshold

Specifies the number of columns that text can be moved to align statements and punctuation, such as declarations, colons, and arrows in named notation. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The default is 16 columns. The full switch name is Format.Alignment_Threshold.

Asm_Listing

Specifies whether an assembly list file should be created for the code generated. This switch should be set only by Rational support personnel. Package Cg_Listing

in !Commands.Internal.Maintenance retrieves these files. The Terminal_Echo flag controls whether these files are also displayed on the screen as they are being generated. The default is false. The full switch name is R1000_Cg.Asm_Listing.

Auto_Login

Specifies that, when an FTP connection is established, FTP should automatically log the user into the remote machine. The default is false. The full switch name is Ftp.Auto_Login.

Closed_Private_Part

Managed by Rational Subsystems™. This switch should not be changed by users.

Comment_Column

Controls in which column the !Commands.Editor.Tab_To_Comment command inserts a comment delimiter. Comments placed in this column remain in that column despite changes to the length of the statement on that same line. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The default is column 1. The full switch name is Format.Comment_Column.

Configuration

Managed by Rational Subsystems. This switch should not be changed by users. The full switch name is Parser.Configuration.

Consistent_Breaking

Controls the formatting of aggregates, discriminants, enumeration types, parameter lists, choice lists, and identifier lists. If this option is true and a list does not fit on a line, every element of the list begins on a new line. The default is true. The full switch name is Format.Consistent_Breaking.

Create_Internal_Links

Sets a Boolean value that specifies whether to create internal links in the set of links for the world for units created in the enclosing world or directories. The default is true. The full switch name is Directory.Create_Internal_Links. (For further information on links, see the Key Concepts in this book.)

Create_Subprogram_Specs

Controls whether specs for library unit subprograms are automatically created when the body is first promoted. The contents of these specs will be created the first time the body is successfully installed. The *with* clause is derived from the *with* clauses in the body. Only those required to promote the spec are included. The default is true. The full switch name is Directory.Create_Subprogram_Specs.

Enable_Deallocation

Specifies whether to enable Unchecked_Deallocation for all access types in units in that library. The default is false. The full switch name is R1000_Cg.Enable-_Deallocation.

Id_Case

Specifies the case of identifiers in Ada units. The options are Lower, Upper, and Capitalized. The default is Capitalized. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect the new units created after the option has been changed. The full switch name is Format.Id_Case.

Ignore_Interface_Pragmas

Causes interface pragmas to be ignored by the semanticist. The default is false. The full switch name is Semantics.Ignore_Interface_Pragmas.

Ignore_Minor_Errors

Sets a Boolean value that specifies whether minor errors should be ignored. Minor errors are those that the *Reference Manual for the Ada Programming Language* defines as illegal but do not affect the semantic validity of the program. An example is illegal declaration order. The default is false. The full switch name is Semantics.Ignore_Minor_Errors.

Ignore_Unsupported_Rep_Specs

Causes errors about unsupported representation specifications to be given as warnings. The default is false. The full switch name is Semantics.Ignore_Unsupported-_Rep_Specs.

Keyword_Case

Specifies the case of keywords as printed by the pretty-printer in Ada units. Options are Lower, Upper, and Capitalized. The default is Lower. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The full switch name is Format.Keyword_Case.

Line_Length

Specifies the number of columns used by the pretty-printer in printing lines in Ada units before wrapping them. The default is 80. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The full switch name is Format.Line_Length.

Major_Indentation

Specifies the number of columns that are to be used for major indentations. The default is 4. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The full switch name is Format.Major_Indentation.

Minor_Indentation

Specifies the number of columns that are to be used for minor indentations. The default is 4. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The full switch name is Format.Minor_Indentation.

Number_Case

Specifies the case to be used in displaying exponentials. The default is uppercase. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The full switch name is Format.Number_Case.

Page_Limit

Controls the page limit for a job executing the unit code that was compiled with this switch. The default is 0. The full switch name is R1000_Cg.Page_Limit.

Password

Specifies the user's remote password for logging into a remote computer over an FTP connection. The default is null. The full switch name is Ftp.Password.

Prompt_For_Account

Causes FTP, when FTP operations are executed, to prompt the user to supply a user account on the connected remote computer. The user is prompted only if the Account switch is null. The default is false. The full switch name is Ftp.Prompt-_For_Account.

Prompt_For_Password

Causes FTP, when FTP operations are executed, to prompt the user to supply a password for the connected remote computer. The user is prompted only if the Account switch is null. The default is false. The full switch name is Ftp.Prompt-_For_Password.

Remote_Directory

Sets a remote directory to which to connect when an FTP connection to a remote computer is being established. The default is null. The full switch name is Ftp.Remote_Directory.

Remote_Machine

Specifies the name of the remote computer to which FTP is to connect. The default is null. The full switch name is Ftp.Remote_Machine.

Remote_Roof

Specifies a directory on a remote computer connected to a local computer through an FTP connection. This directory is an ancestor directory for a group of files that FTP is to transfer. The default is the null string. The full switch name is Ftp.Remote_Roof.

Remote_Type

Specifies the type of the remote computer to which FTP is to connect. The default is Rational. The full switch name is Ftp.Remote_Type.

Require_Internal_Links

Controls whether failure to create internal links (as controlled by the Directory-.Create_Internal_Links switch) generates a warning or an error. The default (true) is to treat failure to generate links as an error and to discontinue the operation. The full switch name is Directory.Require_Internal_Links.

Seg_Listing

Specifies whether a listing of object code is produced. Listings are stored as attributes of the units with which they are associated. Package Cg_Listing in !Commands.Internal.Maintenance retrieves them. The default is false. The full switch name is R1000_Cg.Seg_Listing.

Send_Port_Enabled

Sets a Boolean value that helps diagnose failing FTP transfers by verifying that local and remote machines are using the same connection. The switch should be set to true for transfer of multiple files. The default is false. The full switch name is Ftp.Send_Port_Enabled.

Statement_Indentation

Specifies the number of columns the pretty-printer indents the second and subsequent lines of a statement when the statement has to be broken because it is longer than Line_Length. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The default is 3. The full switch name is Format.Statement_Indentation.

Statement_Length

Specifies the smallest number of columns that the pretty-printer will use to print a statement. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The default is 35. The full switch name is Format.Statement_Length.

Subsystem_Interface

Managed by Rational Subsystems. This switch should not be changed by users. The full switch name is Semantics.Subsystem_Interface.

Target_Key

Managed by Rational Subsystems. This switch should not be changed by users. The full switch name is Policy.Target_Key.

Terminal_Echo

Specifies whether assembly listings are displayed on the terminal as they are generated. The default is false. The full switch name is R1000_Cg.Terminal_Echo.

Transfer_Mode

Sets the mode of an FTP file transfer. Currently, only Stream type is supported. The default is stream. The full switch name is Ftp.Transfer_Mode.

Transfer_Structure

Sets the structure of an FTP file transfer. Currently, only File type is supported. The default is file. The full switch name is Ftp.Transfer_Structure.

Transfer_Type

Sets the type of an FTP file transfer. It can be set to allow text, image, and binary transfers. The default is ASCII. The full switch name is Ftp.Transfer_Type.

Username

Sets the remote username for logging into a remote computer over an FTP connection. The default is the null string. The full switch name is Ftp.Username.

Wrap_Indentation

Specifies the column to which the pretty-printer will extend in wrapping when the current level of wrapping would require statements to be shorter than Statement_Length. Once a Format option has been applied to an Ada unit, changing the option has no effect on that Ada unit, but it does affect new units created after the option has been changed. The default is column 16. The full switch name is Format.Wrap_Indentation.

Commands from package Common

Many of the operations in package !Commands.Common apply to switches. Library switches can be brought into a window with the Switches.Edit procedure. The following procedures from package Common apply to switches. Other operations from package Common that do not apply to switches produce a message to that effect in the Message window when used on switches.

procedure Common.Abandon

Abandons the editing of the switches. The window is removed from the screen. Any changes made to the switches since the last commit operation are lost.

procedure Common.Commit

Commits changes to the switches. Changes to the switches are made in a temporary area of the Environment. To make those changes permanent and to have them take effect, you must commit those changes.

procedure Common.Create_Command

Creates a Command window below the current window. The *use* clause in the Command window, `use Editor, Ada, Switches, Common;`, includes this package, so operations in this package are visible in the Command window without qualification.

procedure Common.Definition

Finds the definition of the selected switch value if that value is a library or library unit. The procedure produces an error for switches that are Booleans, integers, or nonswitch name strings. If the switch is a switch name, a window is brought up with the definition of that object in the window.

procedure Common.Edit

Creates a Command window and places in it the command:

```
Change ("current switch value");
```

where the parameter is the switch value of the switch on which the cursor is located, whether or not there is a selection. Providing a new switch value and promoting the command changes the value of the switch. If the current switch is of Boolean type, the command toggles the value of the switch without creating a Command window.

procedure Common.Elide

Reduces (elides) the number of switches displayed in the window. The window can display all switches in the system (the greatest number displayed) or the nondefault switches in the file (the least number displayed). This procedure reduces the number displayed to the next smaller set. Reducing the number below the nondefault switches has no effect.

procedure Common.Enclosing

Finds the directory or world that contains the switches that are in the current window. If the window contains session switches, the procedure finds the home world for that session. The In_Place parameter specifies whether the library window replaces the switch window.

procedure Common.Expand

Increases (expands) the number of switches displayed in the window. The window can display all switches in the system (the most number displayed) or all nondefault switches in the file (the least number displayed). This procedure increases the number displayed to the next larger set. Increasing the number above all switches in the system has no effect.

procedure Common.Explain

Inserts, below the current switch, an explanation of that switch. If an explanation is already there, this procedure will remove it.

procedure Common.Promote

Commits changes to the switches. Changes to switches are made in a temporary area of the Environment. To make these changes permanent and to have them take effect, you must commit those changes.

procedure Common.Release

Commits changes and ends the editing of the switches. The window is removed from the screen after any changes to the switches are saved.

procedure Common.Revert

Redraws the switches in the current window. If the switches have been changed by another user or program, this procedure redraws the switches to ensure that the image is up to date.

procedure Common.Object.Child

Selects the switch on which the cursor is located. Specifically, if no switch is selected, the procedure selects the switch on which the cursor is located. If a single switch is already selected, the procedure has no effect. If all switches are already selected, the procedure selects the switch on which the cursor is located.

procedure Common.Object.Copy

Copies a highlighted switch from one set of switches to the set of switches on which the cursor is located. If the selected switch and the cursor are both in the same set of switches, the procedure has no effect.

procedure Common.Object.Delete

Deletes the selected switch or the switch on which the cursor is located. A deleted switch assumes a system-defined default value.

procedure Common.Object.First_Child

Selects the first switch of the set.

procedure Common.Object.Insert

Creates a Command window and places in it the command:

```
Insert ("[Processor.]  Switch := Value;");
```

where the parameter must be specified to provide a switch and its value. Specifying a switch and a value for that switch and promoting the command inserts or changes a switch value. Multiple switches can be inserted with one command by separating them with semicolons. This procedure uses the same format as an Options parameter.

procedure Common.Object.Last_Child

Selects the last switch of the set.

procedure Common.Object.Move

Moves highlighted switches from one set of switches to the set of switches on which the cursor is located. If the selected switch and the cursor are both in the same set of switches, the procedure has no effect.

procedure Common.Object.Next

Selects the next switch. If no switch is selected, the procedure selects the switch on which the cursor is located. If all switches are selected, this procedure produces an error.

procedure Common.Object.Parent

Selects the switch on which the cursor is located. If no switch is selected, the procedure selects the switch on which the cursor is located. If a switch is selected, the procedure selects all switches in the set. Otherwise, the procedure has no effect.

procedure Common.Object.Previous

Selects the previous switch. If no switch is selected, the procedure selects the switch on which the cursor is located. If all switches are selected, this procedure produces an error.

procedure Associate

```
procedure Associate (File     : File_Name := "<SELECTION>";
                     Library  : String    := "$";
                     Response : String    := "<PROFILE>");
```

Description

Associates a specified library switch file with a library.

This procedure builds an association between a library switch file and a directory or world. The switches in the library switch file are used for all operations or units within that directory or world. A single library switch file can be associated with several different worlds.

The association is by reference, which means that changes made to the library switch file have an immediate effect on subsequent operations or units in the associated directory or world.

Owner access to the world containing the library is required to execute this procedure successfully. A job running an operation must have read access to the associated switch file.

This procedure is the opposite of the Dissociate procedure.

Parameters

`File : File_Name := "<SELECTION>";`

Specifies the file to be associated with the directory or world. The default is the current selection.

`Library : String := "$";`

Specifies the directory or world to be associated with the switch file. The default is the enclosing directory or world.

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

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

References

procedure Dissociate

---

function Associated

```
function Associated (Library : String := "$") return File_Name;
```

Description

Finds the associated library switch file for the specified directory or world.

This function returns the name of the library switch file that is associated with the specified directory or world. If no library switch file has been associated with the directory or world, the function returns the null string.

Parameters

`Library : String := "$";`

Specifies the directory or world for which the associated library switch file is desired. The default is the enclosing directory or world.

`return File_Name;`

Returns the library switch file associated with the directory or world. If no switch file is associated with the directory or world, the function returns the null string.

Example

A user, Bill, wants to know which library switch file is associated with his home directory. In a Command window, he executes the command:

```
io.echo (switches.associated("!users.bill"));
```

A display results in the Message window, indicating that the library switch file is !Users.Bill.Bills_Switches.

procedure Change

```
procedure Change (Image : Value_Image := ">>SWITCH<<");
```

Description

Changes the value of the selected switch or of the switch on which the cursor is located, if there is no selection.

The procedure changes the value of a switch. The new switch value is provided as a string whose form depends on the type of the switch.

This command is generated in response to using the !Commands.Common.Edit procedure on a switch window, with the current value of the switch replacing the parameter placeholder ">>SWITCH<<".

The Set procedure can be used to change the value of a switch that is not in the switch window.

Parameters

```
Image : Value_Image := ">>SWITCH<<";
```

Specifies the new switch value. The default parameter placeholder ">>SWITCH<<" must be replaced or an error will result.

subtype Composite_Name

```
subtype Composite_Name is String;
```

Description

Defines a name that is the composite name for a switch.

Switches are grouped into *processors*. Each processor has a series of switches for that particular area of the Environment. A switch name is composed of a simple switch name or the processor name and switch name combined. A composite name is composed of the processor name, a period, and the simple switch name. The simple name can be used when it is not ambiguous; otherwise, the composite name must be used.

Example

```
"Session.Cursor_Bottom_Offset"
```

is a composite name for a switch, in which Session is the processor name and Cursor_Bottom_Offset is the simple switch name.

```
"Cursor_Bottom_Offset"
```

is the unambiguous simple name for that switch because there is no other switch named Cursor_Bottom_Offset.

---

procedure Create

```
procedure Create (File      : File_Name  := ">>SWITCH FILE<<";
                  Category  : Character  := 'L';
                  Response  : String     := "<PROFILE>");
```

Description

Creates an empty switch file specified by category.

The kind of switch file created by this procedure is usually for use with libraries (directories or worlds), which is the default. Session switch files also can be created, but they are created automatically when the Edit_Session_Attributes procedure is executed, so the user seldom creates such files.

If the specified file already exists, a new (empty) version is created.

Parameters

`File : File_Name := ">>SWITCH FILE<<";`
Specifies the name of the file to be created. The default parameter placeholder ">>SWITCH FILE<<" must be replaced or an error will result.

`Category : Character := 'L';`
Specifies the kind of switch file to be created. The default is the most common kind, the library switch file. The other kind ('S') is for session switches.

`Response : String := "<PROFILE>";`
Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example

A user named Sue wants to create a switch file with values different from the defaults. This file will be used to change pretty-printer switches in her home directory. She executes the command:

```
switches.create ("sue_switches");
```

in her home directory. Sue now can change the switches and then use the Associate command to associate the new switch file with her home world.

constant Default_File

```
Default_File : constant File_Name := "";
```

Description

Defines a constant that represents the selected, or default, switch file.

If there is no selection or if the selection is not a switch file, the constant represents the switch file associated with the enclosing directory or world. If the enclosing directory or world does not have an associated switch file, the constant represents no switch file.

procedure Define

```
procedure Define (File     : File_Name := ">>SWITCH FILE<<";
                  Response : String    := "<PROFILE>");
```

Description

Creates an empty library switch file.

When the user initially displays the library switch file, a flag appears in the banner (All Switches) indicating that all switches have been set to the system defaults.

If the specified library switch file already exists, a new version is created.

Parameters

`File : File_Name := ">>SWITCH FILE<<";`

Specifies the name of the switch file to be created. The default parameter placeholder ">>SWITCH FILE<<" must be replaced or an error will result.

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

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

procedure Display

```
procedure Display (Names     : Composite_Name  := "@.@";
                   File      : File_Name       := "";
                   Response  : String          := "<PROFILE>");
```

Description

Displays in the log file the specified set of switches from the selected or specified switch.

This procedure displays the switches that match the specified names for the selected or specified switch file. The display is in the log file that is, by default, the current output window. If the default is used for the switch file and no switch file is selected or the enclosing directory or world does not have an associated switch file, an error occurs.

Parameters

`Names : Composite_Name := "@.@";`

Specifies the switches to be displayed from the switch file. The default is all switches.

`File : File_Name := "";`

Specifies the switch file from which to display the switches. The default is the currently selected switch file or the switch file associated with the enclosing directory or world.

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

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

Errors

If the default is used for the switch file and no switch file is selected or the enclosing directory or world does not have an associated switch file, an error occurs.

Example

A user wants to see how her session switches that have "library" as part of their name are set. She executes the command:

```
switches.display ("@.library@", "s_1_switches");
```

The following display, showing all of the specified switches, is the result:

```
-------------------------------------------------------------------------------
!USERS.GZC % SWITCHES.DISPLAY                              STARTED 4:42:26 PM
-------------------------------------------------------------------------------

Contents of Switch File !USERS.GZC.S_1_SWITCHES'V(4)

Processor            Switch                        Type     Value
=========  ============================  =======  =====
Session.   Library_Break_Long_Lines      Boolean  True
Session.   Library_Capitalize            Boolean  True
Session.   Library_Indentation           Integer  2
Session.   Library_Lazy_Realignment      Boolean  True
Session.   Library_Line_Length           Integer  80
Session.   Library_Misc_Show_Edit_Info   Boolean  True
Session.   Library_Misc_Show_Frozen      Boolean  True
Session.   Library_Misc_Show_Retention   Boolean  True
Session.   Library_Misc_Show_Size        Boolean  True
Session.   Library_Misc_Show_Subclass    Boolean  False
Session.   Library_Misc_Show_Unit_State  Boolean  True
Session.   Library_Misc_Show_Volume      Boolean  True
Session.   Library_Shorten_Names         Boolean  True
Session.   Library_Shorten_Subclass      Boolean  True
Session.   Library_Shorten_Unit_State    Boolean  True
Session.   Library_Show_Deleted_Objects  Boolean  False
Session.   Library_Show_Deleted_Versions Boolean  False
Session.   Library_Show_Miscellaneous    Boolean  False
Session.   Library_Show_Standard         Boolean  False
Session.   Library_Show_Subunits         Boolean  True
Session.   Library_Show_Version_Number   Boolean  False
Session.   Library_Std_Show_Class        Boolean  True
Session.   Library_Std_Show_Subclass     Boolean  True
Session.   Library_Std_Show_Unit_State   Boolean  True
Session.   Library_Uppercase             Boolean  False
```

procedure Dissociate

```
procedure Dissociate (Library  : String := "$";
                      Response : String := "<PROFILE>");
```

Description

Removes the association between a directory or world and a library switch file.

This procedure dissociates a library switch file from its previously associated directory or world. If the directory or world does not have an associated switch file, the procedure has no effect.

This procedure is the opposite of the Associate procedure.

Parameters

`Library : String := "$";`

Specifies the directory or world to dissociate from its switch file. The default is the enclosing directory or world.

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

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

References

procedure Associate

procedure Edit

```
procedure Edit (File : File_Name := "");
```

Description

Creates a window in which the set of switches from the specified library switch file can be edited.

The procedure creates a window and displays in it the set of switches from the specified library switch file. If a window already exists with those switches in it, the window is reused. From the window, the switches can be edited with many operations from package !Commands.Common that apply to the window (see the introduction to package Switches for details).

This procedure creates a new window for each set of switches to be edited. To reuse the same window but edit a different set of switches in that window, see the Visit procedure.

If the specified file does not exist, a file of that name is created and the default switches are displayed.

Parameters

```
File :  File_Name := "";
```

Specifies the switch file in which switches are to be edited. The default is the currently selected switch file or the switch file associated with the enclosing directory or world.

References

procedure Visit

procedure Edit_Session_Attributes

```
procedure Edit_Session_Attributes;
```

Description

Creates a window in which the session switches for the current session can be edited.

The procedure creates a window and displays the set of session attributes or session switches for the current session. From the window, the switches can be edited with many operations from package !Commands.Common that apply to the window (see the introduction to package Switches for details).

There is only one set of session switches for a session. This set contains switches that differ from those used and associated with directories or worlds. See the introduction to this package for details about these switches.

subtype File_Name

```
subtype File_Name is String;
```

Description

Defines a name of a switch file.

This name can use any of the wildcards, indirect files, context prefixes, or attributes as long as it is unambiguous. The name can specify a switch file directory or a directory or world. In the latter case, the switch file associated with the directory or world is used.

Further information about general naming, special names, wildcards, context prefixes, and attributes can be found in the Key Concepts in this book.

procedure Insert

```
procedure Insert (Spec : Specification := ">>SWITCHES<<");
```

Description

Inserts one or more switches and switch values into the current set of switches.

The procedure inserts one or more switches in the current set of switches. The new switches and values are displayed in the window. Multiple switches can be added or changed with this procedure by separating the switches with semicolons.

This procedure appears in a Command window in response to the !Commands-.Common.Object.Insert procedure.

Parameters

```
Spec :  Specification := ">>SWITCHES<<";
```

Specifies the switch or switches and their new values to be inserted into the switch file. The default parameter placeholder ">>SWITCHES<<" must be replaced or an error will result.

Example

The command:

```
switches.insert ("cursor_bottom_offset:=50;cursor_top_offset:=50");
```

inserts new values for the two cursor-scrolling offsets.

constant Of_Library

```
Of_Library : constant File_Name := "$";
```

Description

Defines a local constant for the name of the switch file associated with the enclosing library.

Example

The command:

```
switches.set ("directory.create_internal_links:=false",switches.of_library);
```

changes the Create_Internal_Links switch using the constant defining the switches for the current library, Of_Library.

---

constant Of_Session

```
Of_Session : constant File_Name := "<S>";
```

Description

Defines a constant for the name of the switch file associated with the current session.

Example

The command:

```
switches.set ("session.beep_on_errors:=false",switches.of_session);
```

changes the Beep_On_Errors switches for the current session defined by the Of-_Session constant.

---

procedure Set

```
procedure Set (Spec     : Specification := ">>SWITCHES<<";
               File     : File_Name     := "";
               Response : String        := "<PROFILE>");
```

Description

Sets the specified switches and values in the specified switch file.

This procedure changes the value of one or more switches in the specified switch file. The specification defines both the switches to be changed and the new values for those switches.

Parameters

`Spec : Specification := ">>SWITCHES<<";`

Specifies the switches and their new values to be changed in the switch file. The default parameter placeholder `">>SWITCHES<<"` must be replaced or an error will result.

`File : File_Name := "";`

Specifies the switch file to be changed. The default is the selected switch file or the switch file associated with the enclosing directory or world.

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

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

Example

The command:

```
switches.set ("session.beep_on_messages=false","switches_of_session");
```

changes the value of the Session.Capitalize switch from its default value of true to false.

subtype Specification

```
subtype Specification is String;
```

Description

Defines a string that specifies both a switch and a new value for the switch.

This subtype is used in several procedures to specify both a switch to be changed and the new switch value for the switch. The string takes the form of the Options parameter. The option name is the Composite_Name or simple name of the switch. The value is the Value_Image of the switch.

Example

```
"Cursor_Bottom_Offset  := 50"
```

is a specification to change an integer.

```
"Image_Fill_Mode  := True"
```

is a spec to change a Boolean.

```
"Cursor_Bottom_Offset  := 50; Image_Fill_Mode  := True"
```

changes both.

subtype Value_Image

```
subtype Value_Image is String;
```

Description

Defines a string that specifies a value for a switch.

This subtype is used in several procedures and as part of the Specification subtype to specify a new value for a switch.

The form of the string depends on the switch being changed.

procedure Visit

```
procedure Visit (File : File_Name := "");
```

Description

Brings the specified switch file into an existing window where the switches can be edited.

This procedure uses an existing switch window for editing switches and displays the set of switches from the specified file in that window. If a window does not already exist for editing switches, one is created. From the window, the switches can be edited with many operations from package !Commands.Common that apply to the window (see the introduction to package Switches for details).

This procedure does not create a new window for each set of switches to be edited. To create a new window to edit a different set of switches in that window, see the Edit procedure.

Parameters

`File : File_Name := "";`

Specifies the switch file to be edited. The default is the currently selected switch file or the switch file associated with the enclosing directory or world.

References

procedure Edit

---

procedure Write

```
procedure Write (File : File_Name := ">>SWITCH FILE<<");
```

Description

Writes the switches in the current window into the specified file.

This procedure saves the switches that are being edited into a specified file.

Parameters

`File : File_Name := ">>SWITCH FILE<<";`
Specifies the file into which to write the switches. The default parameter placeholder `">>SWITCH FILE<<"` must be replaced or an error will result.

end Switches;

package Xref

This cross-reference package consists of a set of two procedures that generate lists of the Ada units that reference user-selectable Ada constructs in other Ada units. These two procedures, named Xref.Used_By and Xref.Uses, provide analogous functions to the interactive facilities, Show_Usage and Definition, respectively. In the Xref.Used_By procedure, the list of Ada units that reference each occurrence of the user-specified Ada constructs is generated. The Xref.Uses procedure generates the list of Ada units in which each occurrence of the user-specified Ada constructs is defined. For each of these procedures, the line number in which the procedure is defined or referenced is also shown.

The examples in this package reference the Baseball Program in the "Ada Program Modification" section of *Rational Environment Training: Fundamentals*.

procedure Used_By

```
procedure Used_By (List_Of_Names            : String  := "<IMAGE>";
                   Do_Functions             : Boolean := True;
                   Do_Generics              : Boolean := True;
                   Do_Procedures            : Boolean := True;
                   Do_Attributes            : Boolean := False;
                   Do_Record_Components     : Boolean := False;
                   Do_Constants             : Boolean := False;
                   Do_Entries               : Boolean := False;
                   Do_Exceptions            : Boolean := False;
                   Do_Labels                : Boolean := False;
                   Do_Packages              : Boolean := False;
                   Do_Parameters            : Boolean := False;
                   Do_Pragmas               : Boolean := False;
                   Do_Task_Bodies           : Boolean := True;
                   Do_Types                 : Boolean := False;
                   Do_Variables             : Boolean := False;
                   Exclude_References_From  : String  := "";
                   List_File_Name           : String  := "");
```

Description

Produces a list of all the installed or coded Ada units in the entire directory hierarchy that reference the Ada constructs selected by the user in the List_of_Names parameter.

This procedure checks each of the Ada units specified in the List_Of_Names parameter. It then extracts each item that matches one of the Ada constructs specified by the user. The value true is shown in the Do_*Xxx* parameter list. For each item extracted, all references to it are saved. Each item that has at least one reference to it is then output with the name of the item, the name of the Ada unit in which the item is located, and the list of names of the Ada units that reference the item. Following the cross-reference list are two tables that contain the fully qualified name for each of the Ada units.

There are two or three indicators after each of the names of the Ada units. For the Is Referred To By column only, the first is a one- or two-letter flag enclosed in parentheses indicating how the item is used, as shown in Table 10–1. The second is a number enclosed in brackets that refers to one of the two tables of fully qualified names. The rightmost 71 characters of the fully qualified name are displayed. The third set of numbers indicates the line number(s) in which that item is defined or used. This line number is the physical line number within the package in which the definition or usage is made.

The output is directed to the Text_Io.Current_Output device unless a filename is specified in the List_File_Name parameter.

Table 10–1. Xref Flag Definitions

Flag	*Stands for*	*Means*
U	Used	Object is read from or is an in parameter
S	Set	Object is written into or is an out parameter
B	Both used and set	Object is an in/out parameter
UT	Used through	Object is read through pointer
ST	Set through	Object is written through pointer
BT	Both used and set through	Object is an in/out parameter through pointer

An Ada unit can be excluded from the search for references by naming it in the Exclude_Reference_From parameter.

Parameters

```
List_Of_Names : String := "<IMAGE>";
```

Specifies the list of Ada units for which a cross-reference is to be built. The default value is to use the currently selected unit in the current image. For subsystems, this must be the spec view. Refer to Project Management (PM) for more information on subsystems.

```
Do_Functions : Boolean := True;
```

Specifies whether uses of function definitions will be included in the cross-reference.

```
Do_Generics : Boolean := True;
```

Specifies whether uses of generic definitions will be included in the cross-reference.

```
Do_Procedures : Boolean := True;
```

Specifies whether procedure definitions will be included in the cross-reference.

```
Do_Attributes : Boolean := False;
```

This parameter has no effect.

```
Do_Record_Components : Boolean := False;
```

Specifies whether all record component definitions will be included in the cross-reference.

```
Do_Constants : Boolean := False;
```

Specifies whether constants will be included in the cross-reference.

```
Do_Entries : Boolean := False;
```

Specifies whether entry definitions will be included in the cross-reference.

```
Do_Exceptions : Boolean := False;
```

Specifies whether uses of exception names will be included in the cross-reference.

```
Do_Labels : Boolean := False;
```

Specifies whether label definitions will be included in the cross-reference.

```
Do_Packages : Boolean := False;
```

Specifies whether package definitions will be included in the cross-reference.

```
Do_Parameters : Boolean := False;
```
Specifies whether uses of parameters will be included in the cross-reference.

```
Do_Pragmas : Boolean := False;
```
This parameter has no effect.

```
Do_Task_Bodies : Boolean := True;
```
This parameter has no effect.

```
Do_Types : Boolean := False;
```
Specifies whether all type definitions will be included in the cross-reference.

```
Do_Variables : Boolean := False;
```
Specifies whether uses of variables will be included in the cross-reference.

```
Exclude_References_From : String := "";
```
Specifies that all the Ada units listed in this parameter are to be excluded from the search for references to selected elements. The default value is to exclude none.

```
List_File_Name : String := "";
```
Specifies the filename in which to write the output. The default value is Text_Io-.Current_Output.

Restrictions

The Ada units specified in the List_Of_Names parameter must be installed or coded. Only installed or coded Ada units will be checked to see if they reference the specified items.

Example

The command:

```
xref.used_by;
```

produces the following result when run from a window that contains the image of the FORMATTER'SPEC of the Baseball Program contained in the "Ada Program Modification" section of *Rational Environment Training: Fundamentals*:

```
------------------------------------------------------------------------
MAINTENANCE.BASEBALL_SYSTEM.FORMATTER'BODY'V(1)  % USED_BY STARTED Ø7: ...
------------------------------------------------------------------------

ITEM                    DEFINED IN              IS REFERRED TO BY

PRINT_HEADER            FORMATTER[4]4           .BASEBALL_STATISTICS(U)[11]28
PRINT_PLAYER_STATS      FORMATTER[5]5           .BASEBALL_STATISTICS(U)[11]32
PRINT_TEAM_STATS        FORMATTER[6]6           .BASEBALL_STATISTICS(U)[11]36

             FULL NAMES OF "IS REFERRED TO BY" UNITS

2       !USERS.BES.MAINTENANCE.BASEBALL_SYSTEM.BASEBALL_STATISTICS


             FULL NAMES OF "DEFINED IN" UNITS

4     !USERS.BES.MAINTENANCE.BASEBALL_SYSTEM.FORMATTER.PRINT_HEADER
5     !USERS.BES.MAINTENANCE.BASEBALL_SYSTEM.FORMATTER.PRINT_PLAYER_STATS
6     !USERS.BES.MAINTENANCE.BASEBALL_SYSTEM.FORMATTER.PRINT_TEAM_STATS
```

procedure Uses

```
procedure Uses (List_Of_Names              : String  := "<IMAGE>";
                Visible_Declarations_Only  : Boolean := True
                Do_Functions               : Boolean := True;
                Do_Generics                : Boolean := True;
                Do_Procedures              : Boolean := True;
                Do_Attributes              : Boolean := False;
                Do_Record_Components       : Boolean := False;
                Do_Constants               : Boolean := False;
                Do_Entries                 : Boolean := False;
                Do_Exceptions              : Boolean := False;
                Do_Labels                  : Boolean := False;
                Do_Packages                : Boolean := False;
                Do_Parameters              : Boolean := False;
                Do_Pragmas                 : Boolean := False;
                Do_Task_Bodies             : Boolean := True;
                Do_Types                   : Boolean := False;
                Do_Variables               : Boolean := False;
                Exclude_References_To      : String  := "";
                Only_References_To         : String  := "";
                List_File_Name             : String  := "");
```

Description

Produces a list of the installed or coded Ada units that contain the definition of the items selected by the user in the List_Of_Names and Do_*Xxx* parameters.

This procedure checks each of the Ada units specified in the List_Of_Names parameter. It then extracts each item used that matches one of the Ada constructs specified by the user. The value true is shown in the Do_*Xxx* parameter list. For each item, the Uses procedure lists the Ada unit in which it is defined and all the procedures in the Ada units specified in the List_Of_Names parameter that reference the item. The Ada units in this table are referenced by their names. Following the cross-reference list are two tables that contain the fully qualified names for the Ada units.

There are two or three indicators after each of the names of the Ada units. For the Is Referred To By column only, the first is a one- or two-letter flag enclosed in parentheses indicating how the item is used, as shown in Table 10–1 in the Used_By procedure. The second is a number enclosed in brackets that refers to one of the two tables of fully qualified names. The rightmost 71 characters of the fully qualified name are displayed. The third set of numbers indicates the line number(s) in which that item is defined or used. This line number is the physical line number within the unit in which the definition or usage is made.

The output is be directed to the Text_Io.Current_Output device unless a filename is specified in the List_File_Name parameter.

An Ada unit can be excluded from the search for definitions by naming it in the Exclude_Reference_To parameter.

Conversely, the Only_References_To parameter allows the user to specify the Ada units that the user wants searched for the definitions. These last two parameters are mutually exclusive.

Parameters

```
List_Of_Names : String := "<IMAGE>";
```

Specifies the list of Ada units from which the selected objects are to be extracted. The default value is to use the currently selected unit in the current image.

```
Visible_Declarations_Only : Boolean := True;
```

Specifies that only declarations that are exported will be included in the cross-reference.

```
Do_Functions : Boolean := True;
```

Specifies whether uses of functions will be included in the cross-reference.

```
Do_Generics : Boolean := True;
```

Specifies whether generic instantiations will be included in the cross-reference.

```
Do_Procedures : Boolean := True;
```

Specifies whether all procedures called will be included in the cross-reference.

```
Do_Attributes : Boolean := False;
```

This parameter has no effect.

```
Do_Record_Components : Boolean := False;
```

Specifies whether all record components used will be included in the cross-reference.

```
Do_Constants : Boolean := False;
```

Specifies whether all constants used will be included in the cross-reference.

```
Do_Entries : Boolean := False;
```

Specifies whether all entries used will be included in the cross-reference.

```
Do_Exceptions : Boolean := False;
```

Specifies whether uses of exception names will be included in the cross-reference.

```
Do_Labels : Boolean := False;
```
Specifies whether all labels used will be included in the cross-reference.

```
Do_Packages : Boolean := False;
```
Specifies whether all packages used will be included in the cross-reference.

```
Do_Parameters : Boolean := False;
```
Specifies whether all parameters used will be included in the cross-reference.

```
Do_Pragmas : Boolean := False;
```
This parameter has no effect.

```
Do_Task_Bodies : Boolean := True;
```
This parameter has no effect.

```
Do_Types : Boolean := False;
```
Specifies whether all types used will be included in the cross-reference.

```
Do_Variables : Boolean := False;
```
Specifies whether all variables used will be included in the cross-reference.

```
Exclude_References_To : String := "";
```
Specifies a list of Ada units that are not to be included in the output. This parameter cannot be used if the Only_References_To parameter has a nondefault value. The default value is to include all installed or coded Ada units in the entire directory hierarchy.

```
Only_References_To : String := "";
```
Specifies a list of Ada units that are the only ones to be included in the output. This parameter cannot be used if the Exclude_References_To parameter has a nondefault value. The default value is to include all installed or coded Ada units in the entire directory hierarchy.

```
List_File_Name : String := "";
```
Specifies the filename in which to write the output.

Restrictions

The Ada units specified in the List_Of_Names parameter must be installed or coded. Only installed or coded Ada units will be checked to see if they contain the definition of the specified items.

Example

The command

```
xref.uses;
```

produces the following result when run from a window that contains the image of the FORMATTER'BODY of the Baseball Program contained in the "Ada Program Modification" section of *Rational Environment Training: Fundamentals*:

```
------------------------------------------------------------------------

PT_5.BASEBALL_SYSTEM.FORMATTER'BODY'V(1)  % XREF.USES    STARTED 05:59:38 PM
------------------------------------------------------------------------

ITEM                    DEFINED IN              IS REFERRED TO BY

FLOAT_IO                TEXT_IO[5]173           .FORMATTER.FLT_IO(U)[5]7
INTEGER_IO              TEXT_IO[3]143           .FORMATTER.NAT_IO(U)[4]6
NEW_LINE'2              TEXT_IO[21]77           .FORMATTER.PRINT_HEADER(U)[7]23
                                                .PRINT_PLAYER_STATS(U)[8]37
                                                .PRINT_TEAM_STATS(U)[9]48
PUT'4                   TEXT_IO[17]128          .PUT_STATISTIC_VALUES(U)[6]17
                                                .PRINT_PLAYER_STATS(U)[8]32
                                                .PRINT_TEAM_STATS(U)[9]43
PUT_LINE'2              TEXT_IO[22]136          .FORMATTER.PRINT_HEADER(U)[7]26
                                                24
              FULL NAMES OF "IS REFERRED TO BY" UNITS

4         !USERS.PT_5.BASEBALL_SYSTEM.FORMATTER.NAT_IO
5         !USERS.PT_5.BASEBALL_SYSTEM.FORMATTER.FLT_IO
6         !USERS.PT_5.BASEBALL_SYSTEM.FORMATTER.PUT_STATISTIC_VALUES
7         !USERS.PT_5.BASEBALL_SYSTEM.FORMATTER.PRINT_HEADER
8         !USERS.PT_5.BASEBALL_SYSTEM.FORMATTER.PRINT_PLAYER_STATS
9         !USERS.PT_5.BASEBALL_SYSTEM.FORMATTER.PRINT_TEAM_STATS
```

```
              FULL NAMES OF "DEFINED IN" UNITS

3          !IO.TEXT_IO.INTEGER_IO
5          !IO.TEXT_IO.FLOAT_IO
17         !IO.TEXT_IO.PUT'4
21         !IO.TEXT_IO.NEW_LINE'2
22         !IO.TEXT_IO.PUT_LINE'2
```

---

end Xref;

Index

This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

A

B

C

D

E

F

G

H

I

J

K

M

O

P

Q

R

T

U

V

W

X

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043