Rational Environment Reference Manual

Editing Images (EI)

Document Control Number: 8001A-22 (803-002306)

Rev. 1.0, February 1985
Rev. 1.1, June 1985
Rev. 2.0, December 1985
Rev. 3.0, July 1986
Rev. 4.0, 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

How to Use This Book

The Editing Images (EI) book of the *Rational Environment Reference Manual* describes the basic editing commands of the Rational Environment™ that are not dependent on the specific type of image being edited. It contains introductory material on the logic and structure of the basic command set, followed by reference entries for all the packages contained in package Editor. See Editing Specific Types (EST) for more information on type-specific editing operations.

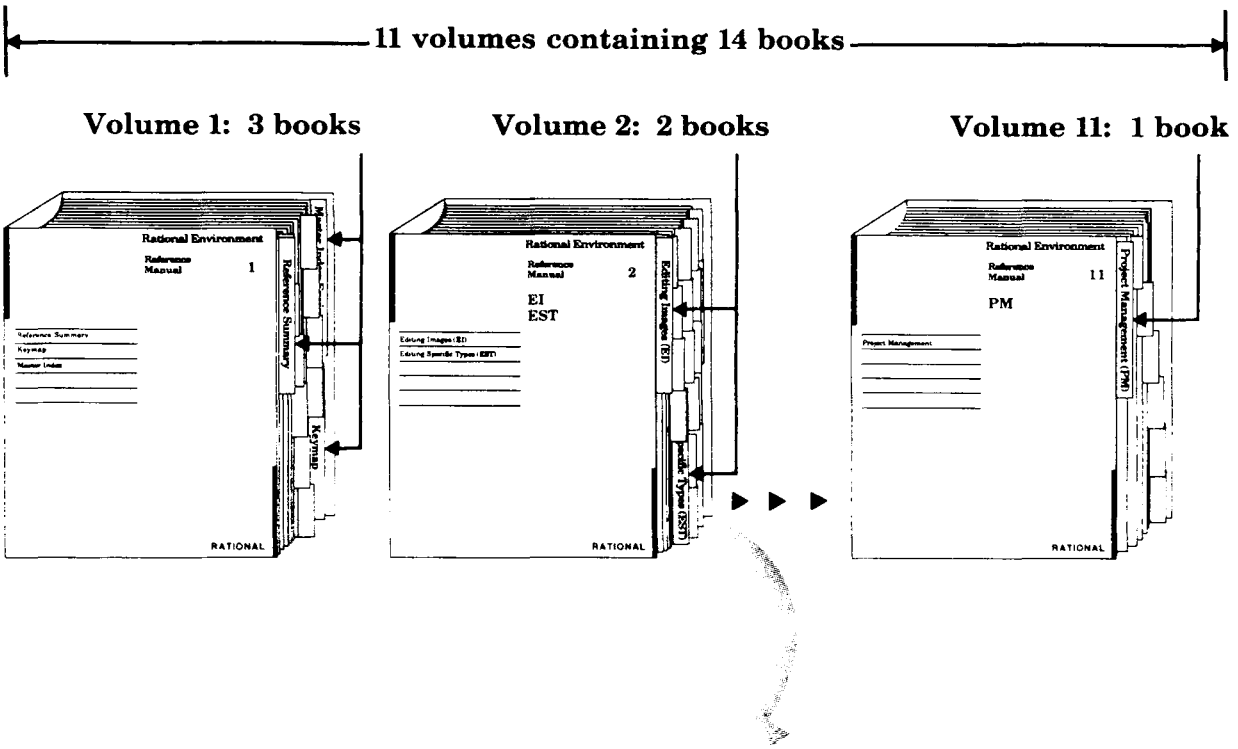
Organization of the Reference Manual

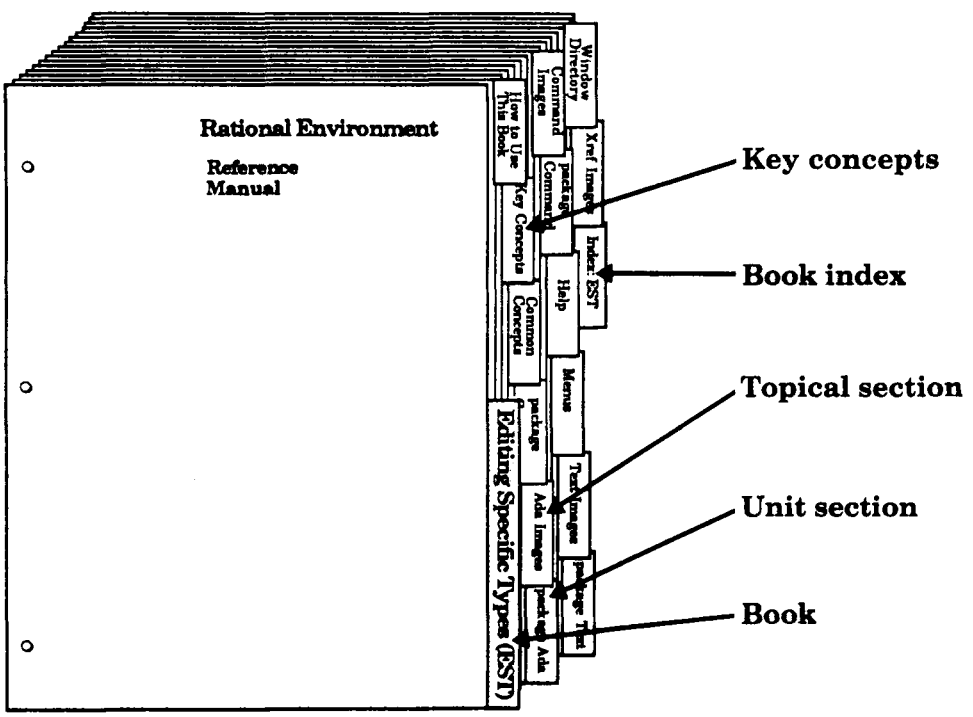
The *Rational Environment Reference Manual* (Reference Manual for brevity) includes the following volumes (see the accompanying illustration):

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:

- **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

The commands in package Editor, common to all images and windows, constitute the means of interacting with the Rational Environment™ at the simplest level. Regardless of the image type—whether Ada®, command, text, or some other type—these operations behave the same. For the most part, these frequently used commands are available at all times for all images, independent of type, and they include procedures for cursor movement, window management, and routine editing. Typically, the commands are bound to a single key or a two-key sequence.

The operations defined in package Editor are used for editing images. There are many of these operations, usually attached or bound to keys. Because these numerous operations would consume many keys, the operations are designed as combinations of *items* and operations, where each item is a key and each operation is a key. Thus, most of these operations involve a two-key sequence. The most common item-operation combinations are bound to single keys.

A detailed discussion of the keymap and the commands that are bound to keys appears in the Rational Environment Keymap (in Volume 1 of the *Rational Environment Reference Manual*). However, because most of the operations defined in this package are normally keys, some discussion of the keymap paradigm follows.

Commands and Keys

Items are the units of text in an image that can be edited; each kind of item is represented on the keyboard by a single key. Operations are the commands that affect cursor position or items; each operation is also represented by a single key. Once the basic key bindings are familiar, any editing operation can be invoked by entering an item key followed by an appropriate operation key.

The first key in an item-operation key sequence acts as a command prefix, indicating that the following keystroke should be interpreted as a command and not as a character to insert in the image. The first key also identifies the item (such as word, line, or region of text) to which the command must apply. For example, Line tells the Rational Editor that a command follows and that it should apply to the current line (the line containing the cursor).

The second key identifies the action that applies to the item. For example, [D] deletes the item. As a rule, the editor ignores the shift key when interpreting the second key; commands are not case-sensitive. For example, [D] is equivalent to [d], and [?] is equivalent to [/]. However, the arrow, or cursor, keys have different meanings for the shifted or not-shifted versions of the keys.

A command is entered by pressing the item key, releasing it, and pressing the operation key. Some commands use [Control], [Meta], or [Shift] (or combinations of the three) in combination with another key. These three keys are called *modifier keys*. [Control], [Meta], and [Shift] should be pressed and held down while the other key is pressed, shown as [Control][F10], for example, in this documentation. (Keys that are pressed and released sequentially are shown as [Window] - [↑].)

More information about keys and the commands that are bound to keys can be found in the Rational Environment Keymap (in Volume 1 of the *Rational Environment Reference Manual*).

Cursor Movement

The position of the cursor determines where keyboard entries appear in the image. The cursor position is often important for specifying the location or affected area for editing operations.

An image has topmost and leftmost boundaries (row 1 and column 1). However, there are no bottommost or rightmost boundaries. This defines what is called the *quarter plane* in which the cursor can move.

Except for the top and left boundaries, cursor movement in an image is virtually unlimited. The cursor can move to any location, regardless of whether text is present.

The cursor can be controlled in different ways. Planar movement is absolute movement (up, down, left, and right) anywhere on the screen or in an image. Relative movement is movement by item—whether word, line, or window—and depends on the cursor's current context.

The cursor can also be moved to a specific image position that has been previously saved as a *mark* (see package Mark) or to a specific portion of text previously designated as a *region* for editing purposes (see package Region).

The sections that follow explain these several types of cursor movement.

Planar Movement

Planar, or absolute, cursor movement is movement by fixed size or length. Commands for planar movement include up, down, left, and right. They are defined in packages Cursor, Image, and Screen, as follows:

- Package Cursor contains procedures for movement within a window. These commands are bound to the arrow keys, which move the cursor in the indicated direction. Cursor movement, however, is restricted by the boundaries of the current window. Because window boundaries block further cursor movement, the Rational Editor scrolls the image to keep the cursor visible in the window.
- Package Image scrolls the contents of the window. These commands are bound to the arrow keys with the Image prefix. They scroll the image in the indicated direction to the next page (windowful). The amount of the previous page that stays on the screen can be specified with a switch. There are also single-key bindings for scrolling the image; see the Rational Environment Keymap (in Volume 1 of the *Rational Environment Reference Manual*).
- Package Screen contains procedures for unrestricted movement across the screen. These commands are also bound to the arrow keys with Control and Shift as prefixes. They move the cursor in the indicated direction but ignore window boundaries.

Stream Operations

The Rational Editor provides stream operations as well as planar operations. In stream operations, the editor assumes that lines are separated by a single character and that the end of one line is contiguous to the next. The Environment provides stream operations in the following packages and procedures:

- Package Char: procedures Delete_Forward, Delete_Backward
- Package Cursor: procedures Forward, Backward
- Package Word: procedures Delete_Forward, Delete_Backward

Relative Movement

Relative movement differs from regular cursor movement in that its direction and extent depend on the location of the cursor and the item to which the procedure applies. Operations for moving the cursor relative to the current item are defined in the following packages:

- Package Image
- Package Line
- Package Window
- Package Word

These packages define commands for moving to the *next* or *previous* words, lines, and windows. They also provide commands for moving to the *beginning of* and *end of* words, lines, windows, and images.

The Rational Editor provides two commands, the Cursor.Next and Cursor.Previous procedures, for moving between prompts and underlines. These procedures have parameters that allow users to specify their preferences.

Marks

Marks are remembered image positions. They can facilitate movement within an image or between images. They can also mark a location in an image for later reference. Operations for setting and manipulating marks are defined in package Mark.

The editor uses the mark stack to keep track of mark positions. There is one mark stack for all images. Pushing the stack sets a mark, and the Top procedure retrieves—moves the cursor to—the most recently referenced mark. The Next and Previous procedures cycle through the stack, wrapping at the top and bottom. The Copy_Top procedure makes another copy of the top item on the stack on the top of the stack. The Delete_Top procedure removes the top item from the stack. The Swap procedure swaps the top two items on the stack. The Rotate procedure moves the bottom of the stack to the top of the stack.

It is possible to set a mark anywhere in any image. The editor remembers the position as an absolute image position, effectively a set of line and column coordinates on the quarter plane. These coordinates do not change to adjust for insertions or deletions.

Search and Replace

The Rational Editor provides operations for searching forward and backward and for replacing strings of text. Search and replace operations are defined in package Search. Because of the special nature of these commands, they have single-key bindings that prompt for necessary parameters.

Search and replace procedures can be used for regular expression matching. The Wildcard parameter allows strings to be interpreted as regular expressions. Regular expressions allow the user to search for strings matching various combinations of characters, Ada delimiters, and so on.

By default, each command prompts with the most recently used strings as default parameters. The user can replace the prompt by typing over it. Editing and reusing portions of the prompt is done by using the Set.Designation_Off procedure to convert the prompt to text.

Editing Operations

Text-editing operations make up the majority of procedures in package Editor. Despite the diversity and extent of the command set, however, the number of keys the user needs to learn is relatively small because the bindings for specific items and the bindings for specific operations never change, regardless of how the items and operations combine to form a particular procedure. [Line], for example, always indicates a line, whether the operation is delete, capitalize, or transpose. Similarly, [<] always indicates lowercase, whether the procedure is applied to a character, word, line, or region of text.

The text-editing operations discussed in this section apply uniformly to all items as defined in the following packages:

- Package Char
- Package Line
- Package Region
- Package Word

Editing Text

No procedure is needed for inserting characters. The Rational Editor normally inserts characters at the cursor, making room (if necessary) between characters already on the line. In overwrite mode (overwrite mode or insert mode can be changed with the Set.Insert_Mode procedure), the editor replaces existing characters with characters entered from the keyboard.

The editor can be set to break lines automatically at a specified fill column as text is entered by putting a window in fill mode (fill mode can be changed with the Set.Fill_Mode procedure).

The editor relies on the cursor to indicate which character, word, or line should be deleted. Depending on the operation, the item is deleted entirely, deleted from the cursor forward, or deleted from the cursor backward. The Region.Delete procedure deletes the current region regardless of cursor position.

Text-changing operations transpose the position of items or change the case of the alphabetic characters in the specified item. The case-changing procedures always affect the image from the cursor to the end of the specified item. The Transpose procedure assumes that the cursor is on the second of the items to be transposed.

Retrieving Text

Areas of text that are deleted (other than individual character deletions or multiple character deletions made using character delete operations with arguments greater than 1) are saved in a *hold stack*, which is used for cutting, pasting, and copying areas of text. The hold stack is used with the procedures in package Hold_Stack.

For each session, there is only one hold stack for all images. It is used to store deleted words, lines, and regions. The Push procedure also puts the current selection (see packages Common.Object (EST) and Hold_Stack) on the hold stack.

Lines and Tabs

Procedures for inserting and joining lines are defined in package Line. The cursor can be moved by lines, and lines can also be deleted, centered, copied, indented, or have their case changed with procedures in this package.

Procedures for tabbing are defined in package Char. The cursor can be moved by tab stop with the Tab_Forward, Tab_Backward, or Tab_To_Comment procedures.

Note: Lines and tabs are blank spaces inserted in the image. They are not special characters. The Tab_Forward procedure inserts blanks in the image up to the next tab stop. The Tab_Backward procedure, on the other hand, deletes blanks and characters backward to the previous tab stop. The Tab_To_Comment procedure tabs to the comment column specified in the Library switch file.

Procedures for setting or changing tab stops are defined in package Set. For each session, tab stops can be set and removed in particular columns with the Tab_On and Tab_Off procedures, respectively. For all images, the default tab width can be changed with the Tab_Width procedure. Procedures that place, remove, and modify tab stops are defined in package Set.

Selecting Text

Selection commands are defined in packages Region and Common.Object (EST). Selections can be used for editing, moving, and copying text. Selections made using package Region contain arbitrary strings of text, whereas selections made using package Object are hierarchically related (word, line, paragraph, or the entire image).

For information on using text selections with the hold stack, see "Retrieving Text," above.

Window Management

Commands for managing or otherwise manipulating windows are defined in package Window. All window commands use Window as a prefix.

The Window.Directory procedure displays a window, called the *Window Directory*, that contains a list of all currently active images and indicates their names, sizes, types, times of last modification, and whether they have been modified since last committed. From the Window Directory window (whose image type is windows), it is possible to move to another window by using the !Commands.Common.Definition command and to perform other operations, such as committing the window's contents.

The Environment handles routine window management, making a number of decisions about placing and removing windows. However, several procedures defined in package Window give the user control over several aspects of the screen.

Windows can be copied (split to support two views of an image) and joined (expanded to occupy the space of the next window). They can be deleted altogether or removed from the screen. They can be transposed (the position of one window is switched with that of another). The procedure names are consistent with those for text editing; the actual effects of the commands are analogous but not always identical.

Screen Management

A facility for managing screens is provided by package Screen. This package provides a screen stack to keep track of a screen of windows, including which windows were on the screen, their sizes, and the cursor locations in those windows, so that they can be retrieved later in the same session. Changes made to the screen between the time a screen is saved and retrieved are reflected on the screen upon its retrieval. The user can use the Push procedure to add a screen to the top of the stack and later use the Top procedure to retrieve the most recently referenced screen. The Next and Previous procedures cycle through the stack of screens, wrapping at the top and bottom. The Copy_Top procedure copies the top screen of the stack on top of the stack. The Delete_Top procedure removes the top screen from the stack. The Swap procedure swaps the top two screens on the stack. The Rotate procedure moves the screen on the bottom of the stack to the top of the stack.

Macros

A *macro* is a sequence of editor operations that can be invoked with a single command. Macros can be saved for immediate and temporary use. Macros are defined with the Macro.Start and Macro.Finish procedures and are invoked with the Macro.Execute procedure. Frequently used macros can also be bound to keys and stored permanently.

A new macro definition always replaces an existing definition but does not affect macros already bound to keys. The current keyboard macro can be invoked while a new macro is being defined.

Macros are intended for simple, repetitive editing operations. They behave as operations that have no parameters, no local declarations, and no control structures. For more complex editing operations, such as those requiring parameter passing, Ada procedures can be used.

Procedures for defining, executing, and saving keyboard macros are defined in package Macro.

Session Switches

The behavior of many of the facilities provided by the Rational Editor can be tailored by session switches in addition to the commands provided in package Editor. For more information on session switches, see Session Switches in the Session and Job Management (SJM) book of the *Rational Environment Reference Manual.*

The kinds of editor behavior that can be tailored include:

- Setting tab stops.
- Defining when to sound the terminal bell.
- Determining when to scroll images.
- Setting the cursor location after transpose operations.

- Setting the default fill and insert/overwrite modes.
- Defining the directory from which key binding and macro definitions are taken.
- Defining word breaks and prompt delimiters.
- Tailoring the display of windows.

package Editor

The Rational Environment's basic editing operations are defined in package Editor. These operations apply to all objects regardless of type and thus are primarily text-editing operations. They should be used in conjunction with the type- and context-sensitive procedures described in the Editing Specific Types (EST) book of the *Rational Environment Reference Manual.*

The subpackages are named and organized by the objects or items to which editing operations apply, as follows:

- Package Char: Editing operations for single characters.
- Package Cursor: Operations for cursor movement within an image (up, down, left, and right); operations for moving the cursor by designation (prompt or error).
- Package Hold_Stack: Operations for manipulating the hold stack and for retrieving deletions.
- Package Image: Operations for scrolling the image.
- Package Key: Procedures for defining and saving new or custom key bindings.
- Package Line: Editing operations for lines; operations for moving the cursor by relative line position within a window and for inserting and joining lines.
- Package Macro: Operations for defining, executing, saving, and binding macros.
- Package Mark: Operations for marking locations in an image for future reference or for moving between images.
- Package Region: Operations for defining, editing, and manipulating regions of text.
- Package Screen: Operations for moving the cursor on the screen regardless of window boundaries; facilities for dumping and refreshing screen contents; facilities for manipulating the screen stack.
- Package Search: Procedures for searching and replacing strings of text.
- Package Set: Procedures for modifying default editor parameters.
- Package Window: Operations for managing or manipulating windows and frames; operations for moving between windows.

- Package Word: Editing operations for words; operations for moving the cursor by relative word position within a window.

```
procedure Alert;
```

Rings the terminal bell.

```
procedure Noop;
```

Performs no operation; can be used to define a key to do nothing.

```
procedure Quit (Ignore_Changes : Boolean := False);
```

Ends the current session.

If the Ignore_Changes parameter is false (the default), the Quit procedure ends the current session only if there are no new or edited images that have not been committed. If there are images that have not been committed, the Window Directory can be used to find these images and to commit them before quitting. See the Editing Specific Types (EST) book of the *Rational Environment Reference Manual* for more information on the editing operations available on the Window Directory.

If the Ignore_Changes parameter is true, the Quit procedure does not warn whether an image needs committing. In that case, it simply ends the current session and ignores any changes to uncommitted images.

Note that the editor treats I/O windows with jobs requesting input running in them as uncommitted images. You cannot log off while such jobs are running without terminating them or ignoring changes.

package Char

```
package Char is
   procedure Capitalize        (Repeat    : Integer      := 1);
   procedure Delete_Backward   (Repeat    : Integer      := 1);
   procedure Delete_Forward    (Repeat    : Integer      := 1);
   procedure Delete_Next       (Repeat    : Integer      := 1);
   procedure Delete_Previous   (Repeat    : Integer      := 1);
   procedure Delete_Spaces     (Remaining : Natural      := 1);
   procedure Insert_String     (Value     : String);
   procedure Insert_Character  (Repeat    : Integer      := 1;
                                Value     : Character);
   procedure Lower_Case        (Repeat    : Integer      := 1);
   procedure Quote;
   procedure Tab_Backward      (Repeat    : Integer      := 1);
   procedure Tab_Forward       (Repeat    : Integer      := 1);
   procedure Tab_To_Comment;
   procedure Transpose         (Offset    : Integer      := 1);
   procedure Upper_Case        (Repeat    : Integer      := 1);
end Char;
```

Description

Package Char contains character-editing operations. The cursor always rests between two characters or blanks. However, the cursor appears to be on the second character of the two characters it rests between. For example, in the word *and*, when the cursor appears to be on *n*, it is actually located between the *a* and the *n*. All character-editing operations affect the characters before or after the cursor Repeat times.

The case-changing operations move the cursor forward Repeat positions. They check for an alphabetic character in each position and, if they find one, change its case.

The Tab_Forward procedure inserts blanks forward to the next tab stop. The Tab_Backward procedure deletes all characters and blank spaces back to the previous tab stop.

The Insert procedures insert the value specified in the Value parameter in the image at the cursor.

By default, characters entered at the keyboard are inserted in the image. If the editor must make room for a character in the line, it moves any characters that lie to the right of the cursor. Conversely, when a character is deleted, the editor shifts characters to occupy the emptied position.

In overwrite mode, characters in the image are not moved to make room for new insertions. Instead, a character entered from the keyboard replaces the character under the cursor. (See the Set.Insert_Mode procedure.)

Use the Quote procedure to insert special characters (such as control characters) in the image. Special characters appear in the image as highlighted characters. For example, the Control-L character appears as a highlighted L in the image.

The Delete_Spaces procedure removes blanks between the cursor and the next non-blank character.

```
procedure Capitalize (Repeat : Integer := 1);
```

Capitalizes the words in the next Repeat characters.

If the cursor is within a word, the procedure capitalizes the "word" beginning at the cursor. The cursor is left after the last word capitalized.

```
procedure Delete_Backward (Repeat : Integer := 1);
```

Deletes Repeat characters to the left of the cursor.

When the cursor reaches the beginning of the line, the line becomes joined with the line before it. The cursor continues deleting on that line.

```
procedure Delete_Forward (Repeat : Integer := 1);
```

Deletes Repeat characters to the right of the cursor.

When the cursor reaches the end of the line, the line becomes joined with the line after it. The cursor continues deleting on that line. The cursor remains in its original position.

```
procedure Delete_Next (Repeat : Integer := 1);
```

Deletes Repeat next characters but does not delete beyond the current line.

If Repeat is greater than the number of characters on the line, the procedure deletes to the end of the line and stops. The cursor remains in its original position.

```
procedure Delete_Previous (Repeat : Integer := 1);
```

Deletes Repeat previous characters but does not delete beyond the current line.

If Repeat is greater than the number of characters on the line, the procedure deletes to the beginning of the line and stops.

```
procedure Delete_Spaces (Remaining : Natural := 1);
```

Deletes blanks and joins lines from the cursor to the next nonspace character (in both directions), leaving Remaining spaces.

```
procedure Insert_Character (Repeat : Integer := 1;
                           Value  : Character);
```

Inserts the Value character Repeat times at the cursor.

After the operation, the cursor is located after the rightmost character inserted.

```
procedure Insert_String (Value : String);
```

Inserts Value at the cursor.

After the operation, the cursor is located after the rightmost character inserted.

```
procedure Lower_Case (Repeat : Integer := 1);
```

Converts Repeat characters to lowercase.

The procedure moves forward Repeat spaces and, where it finds an uppercase alphabetic character, converts the character to lowercase.

```
procedure Quote;
```

Quotes a special character, such as a control character, into the image.

This procedure must be used for each special character inserted into the image. The procedure cannot be called from a Command window but must be invoked through a key binding; use the Insert_Character procedure or the Insert_String procedure to insert special characters from a Command window or program.

```
procedure Tab_Backward (Repeat : Integer := 1);
```

Deletes backward to the previous tab stop Repeat times.

This procedure affects only the current line.

```
procedure Tab_Forward (Repeat : Integer := 1);
```

Inserts blanks to the next tab stop Repeat times.

This procedure affects only the current line.

```
procedure Tab_To_Comment;
```

Moves the cursor to the first character of the text in the comment, if there is a comment on the current line.

If there is no comment on the line and no text currently located at the comment column (set by the Comment_Column library switch), this procedure inserts the comment characters (--) and leaves the cursor at the first character position for the comment. Otherwise, if there is text at the comment column, the procedure moves the cursor to the end of the text, inserts the comment characters (--), and leaves the cursor after the inserted characters.

```
procedure Transpose (Offset: Integer := 1);
```

Transposes the Offset characters before and after the cursor.

The cursor actually appears on the second character of the two characters it rests between. Thus, this procedure transposes the character on which the cursor is located and the character before it. The procedure transposes characters on the same line. The cursor position that results depends on the value of the Cursor_Transpose_Moves session switch. If the value is true, the cursor is moved to the right one character position. If it is false, the position of the cursor is not changed. For more information on session switches, see SJM, Session Switches.

```
procedure Upper_Case (Repeat : Integer := 1);
```

Converts Repeat characters to uppercase.

The procedure moves forward Repeat spaces and, where it finds a lowercase alphabetic character, converts the character to uppercase.

RATIONAL

package Cursor

```
package Cursor is
  procedure Down     (Repeat     : Integer := 1);
  procedure Left     (Repeat     : Integer := 1);
  procedure Right    (Repeat     : Integer := 1);
  procedure Up       (Repeat     : Integer := 1);
  procedure Forward  (Repeat     : Integer := 1);
  procedure Backward (Repeat     : Integer := 1);
  procedure Next     (Repeat     : Integer := 1;
                      Prompt     : Boolean := True;
                      Underline  : Boolean := True);
  procedure Previous (Repeat     : Integer := 1;
                      Prompt     : Boolean := True;
                      Underline  : Boolean := True);
end Cursor;
```

Description

The Up, Down, Left, and Right procedures move the cursor in the indicated direction Repeat times.

The Next and Previous procedures move the cursor by prompts or underlines.

The Forward and Backward procedures move the cursor left and right, continuing cursor movement on the next or previous line when the cursor reaches the end or beginning of the current line.

Cursor movement is restricted by window boundaries with commands in this package. (Commands in package Screen can be used to move the cursor across window boundaries.) At window boundaries, the editor keeps the cursor visible by scrolling the image.

The cursor highlights a unique location in each image, indicating to the editor where to place keyboard insertions. The cursor is a point of reference for focusing the attention of the Environment, specifying the domain for any command in progress or the object to which an operation applies. The cursor actually appears on a character but represents the position between the character it is on and the character before it.

```
procedure Backward (Repeat : Integer := 1);
```

Moves the cursor to the left Repeat characters.

When the cursor is at the beginning of the current line, this procedure moves the cursor to the position after the last character on the previous line. Negative values move the cursor in the opposite direction, to the right.

```
procedure Down (Repeat : Integer := 1);
```

Moves the cursor down Repeat lines.

Negative values move the cursor in the opposite direction.

```
procedure Forward (Repeat : Integer := 1);
```

Moves the cursor to the right Repeat characters.

When the cursor is one position past the last character on the current line, this procedure moves the cursor to the first character on the next line. Negative values move the cursor in the opposite direction, to the left.

```
procedure Left (Repeat : Integer := 1);
```

Moves the cursor Repeat columns to the left, stopping at the beginning of the line.

Negative values move the cursor in the opposite direction, to the right.

```
procedure Next (Repeat     : Integer := 1;
                Prompt     : Boolean := True;
                Underline  : Boolean := True);
```

Moves the cursor forward to the next underline or prompt.

Negative values for the Repeat parameter move the cursor in the opposite direction. If the Prompt parameter is true (the default), the procedure looks for the next prompt. If the Underline parameter is true, the procedure looks for the next underline. When both parameters are true, the procedure looks for the next occurrence of either a prompt or an underline; if both are false, it does nothing.

```
procedure Previous (Repeat     : Integer := 1;
                    Prompt     : Boolean := True;
                    Underline  : Boolean := True);
```

Moves the cursor backward to the previous underline or prompt.

Negative values for the Repeat parameter move the cursor in the opposite direction. If the Prompt parameter is true (the default), the procedure looks for the previous prompt. If the Underline parameter is true, the procedure looks for the previous underline. When both parameters are true, the procedure looks for the previous occurrence of either a prompt or an underline; if both are false, it does nothing.

```
procedure Right (Repeat : Integer := 1);
```

Moves the cursor Repeat columns to the right on the current line.

Negative values move the cursor in the opposite direction, stopping at the beginning of the line.

```
procedure Up (Repeat : Integer := 1);
```

Moves the cursor up Repeat lines.

Negative values move the cursor in the opposite direction.

---

package Hold_Stack

```
package Hold_Stack is
  procedure Copy_Top;
  procedure Delete_Top;
  procedure Next          (Repeat : Integer := 1);
  procedure Previous      (Repeat : Integer := 1);
  procedure Push          (Repeat : Integer := 1);
  procedure Rotate        (Repeat : Integer := 1);
  procedure Swap;
  procedure Top;
end Hold_Stack;
```

Description

Package Hold_Stack provides a mechanism for recovering deletions. The Rational Editor saves the 100 most recent deletions (larger than a single character) in the hold stack. Deleted selections of any size are also saved. Items in the stack can be retrieved and inserted at the cursor with the Top, Next, or Previous procedure.

An item retrieved from the hold stack is displayed and treated as a selection.

The Push procedure puts the current selection in the hold stack. The Top procedure inserts the most recent item back into the image at the cursor. The Next and Previous procedures provide a mechanism for moving through the items in the stack. They replace the selection at the cursor with the next or previous item in the stack. At the bottom of the stack, Next wraps to the top. At the top of the stack, Previous wraps to the bottom. The Copy_Top procedure makes a copy of the top item of the stack and puts it on the top of the stack. The Delete_Top procedure removes the top item from the stack. The Rotate procedure takes the bottom item of the stack and places it on the top of the stack. The Swap procedure swaps the top two items on the stack.

```
procedure Copy_Top;
```

Copies the top item of the hold stack onto the top of the hold stack.

The result is that there are two copies of the same item on the top of the hold stack.

```
procedure Delete_Top;
```

Removes the top item from the hold stack.

The item next to the top of the hold stack becomes the top item of the hold stack.

```
procedure Next (Repeat : Integer := 1);
```

Retrieves the Repeat item from the hold stack and copies it at the current cursor location.

At the bottom of the stack, this procedure wraps to the top.

```
procedure Previous (Repeat : Integer := 1);
```

Retrieves the previous Repeat item from the hold stack and copies it at the current cursor location.

At the top of the stack, this procedure wraps to the bottom.

```
procedure Push (Repeat : Integer := 1);
```

Pushes the current selection onto the hold stack Repeat times.

```
procedure Rotate (Repeat : Integer := 1);
```

Takes the Repeat number of items from the bottom of the stack and places them on the top of the hold stack in order.

```
procedure Swap;
```

Swaps the top two items on the hold stack.

The item next to the top of the hold stack becomes the top item of the hold stack. The original top item becomes the item next to the top of the hold stack.

```
procedure Top;
```

Retrieves the top item in the hold stack, leaving that item on the hold stack.

The retrieved item is copied into the current cursor location.

RATIONAL

package Image

```
package Image is
  procedure Up           (Repeat : Integer  := Ø);
  procedure Down         (Repeat : Integer  := Ø);
  procedure Left         (Repeat : Integer  := Ø);
  procedure Right        (Repeat : Integer  := Ø);
  procedure Find         (Name   : String);
  procedure Beginning_Of (Offset : Natural  := Ø);
  procedure End_Of       (Offset : Natural  := Ø);
end Image;
```

Description

Package Image includes operations for finding and scrolling images. The Find procedure searches for an image of a given name in the Window Directory and displays it on the screen. The Up, Down, Left, and Right procedures scroll the image Repeat lines or columns in the indicated direction. The default (Repeat = 0) scrolls the image one full window. When scrolling a full window, the editor keeps a portion of the previous view to preserve context (how much it keeps can be specified with switches).

The Beginning_Of and End_Of procedures move the view as indicated but use Offset to adjust the new cursor position by Offset lines from the beginning or end of the image.

```
procedure Beginning_Of (Offset : Natural := Ø);
```

Moves the cursor to the beginning of the image.

This procedure uses Offset to adjust the new cursor position by Offset lines from the beginning of the image.

```
procedure Down (Repeat : Integer := Ø);
```

Scrolls the image forward Repeat lines.

When Repeat = 0, the procedure scrolls one full window. If Repeat is a negative number, it scrolls down Repeat lines.

```
procedure End_Of  (Offset : Natural := Ø);
```

Moves the cursor to the end of the image.

This procedure uses Offset to adjust the new cursor position by Offset lines from the end of the image.

```
procedure Find (Name : String);
```

Searches the Window Directory for the specified image, or for an image that contains a substring of Name in its name, and brings it onto the screen.

If the null string is supplied as Name, the procedure brings back to the screen the last image the Environment removed from the screen (this does not include images explicitly removed by users either by deleting them or marking them for replacement).

```
procedure Left (Repeat : Integer := Ø);
```

Scrolls the image left Repeat columns.

When Repeat = 0, the procedure scrolls one full window. If Repeat is a negative number, it scrolls right. If the first character of the image is already in the first character position of the window, the procedure does nothing.

```
procedure Right (Repeat : Integer := Ø);
```

Scrolls the image right Repeat columns.

When Repeat = 0, the procedure scrolls one full window. If Repeat is a negative number, it scrolls left.

```
procedure Up (Repeat : Integer := Ø);
```

Scrolls the image back Repeat lines.

When Repeat = 0, the procedure scrolls one full window. If Repeat is a negative number, it scrolls down. If the first line of the image is already at the top of the screen, the procedure does nothing.

---

package Key

```
package Key is
  procedure Define (Key_Name      : String  := ">>KEY NAME, e.g. CM_F1<<";
                    Command_Name  : String  := ">>COMMAND NAME<<";
                    Prompt        : Boolean := False);
  procedure Name   (Key_Code      : String  := "");
  procedure Save;
  procedure Prompt (Key_Code      : String  := "");
end Key;
```

Description

Use the procedures in package Key to bind any procedure from !Commands or any fully qualified Ada procedure call to a key. These bindings are in effect until the user logs off.

For example, function key names are defined as follows:

F1–F20	Function keys—for example, F10 is the Definition key.
S_*key*	The Shift key pressed along with another key, nonalphanumeric—for example, S_F10.
C_*key*	The Control key pressed along with another key—for example, C_G or C_F10.
M_*key*	The Meta key pressed along with another key—for example, M_F10.
CS_*key*	The Control and Shift keys pressed along with another key—for example, CS_F10.
CM_*key*	The Control and Meta keys pressed along with another key—for example, CM_F10.
MS_*key*	The Meta and Shift keys pressed along with another key—for example, MS_F10.
CMS_*key*	The Control, Meta, and Shift keys pressed along with another key—for example, CMS_F10.

To determine the key name for a key, use the !Commands.What.Does or What.Key procedure.

Another way to change the key bindings is to create a permanent local key binding procedure. This is outlined below.

At login, the Environment assigns the key bindings defined in the !Machine.Editor_Data.Rational_Commands procedure, which globally assigns commands to key bindings for the entire system. This procedure applies to the Rational Terminal.

Other procedures exist for each of the types of terminals that can be specified at login.

Users who want to create their own local key bindings should build in their home world a procedure called Rational_Commands that uses the same form as the default procedure, !Machine.Editor_Data.Rational_Commands. A template for this procedure, called !Machine.Editor_Data.Rational_User_Commands, contains the case statements and the proper context clauses. Like the default procedure, this local procedure should contain a case statement; it needs only those cases defined that differ from the cases of the default procedure.

To create a local keymap, create a Rational_Commands procedure in your home world and copy the contents of the !Machine.Editor_Data.Rational_User_Commands procedure into it. Modify the case statement as desired, install the procedure, quit, and then log in again for the new key bindings to take effect.

The Rational_Commands procedure applies specifically to the Rational Terminal. Other terminals that the Environment supports need similar procedures with different names for their key bindings. The directory !Machine.Editor_Data contains procedures and templates for each of the terminals supported by the Environment.

```
procedure Define (Key_Name      : String  := ">>KEY NAME, e.g. CM_F1<<";
                  Command_Name  : String  := ">>COMMAND NAME<<";
                  Prompt        : Boolean := False);
```

Binds a command to a key.

The Key_Name parameter specifies the key to which the command should be bound. To find out the name of a key, use the !Commands.What.Does or What.Key procedure. The Command_Name parameter specifies the command to be bound to Key_Name—for example, Text.Create. The Prompt parameter specifies whether a Command window should be brought up containing the command.

The Command_Name parameter either can take simple names (if they are in the current context) or can be resolved via your searchlist when it is defined. The key is bound to the object itself, rather than the name. Thus, if there is an object in the current context with the same simple name as Command_Name, the system will retrieve the correct one.

To bind a procedure that has parameters and is not in your current searchlist, you must enclose the name in quotes. For example, to call a procedure called Phone in !Users.John, with a parameter requesting the name of a person in the phone list, you would enter:

```
Key.Define (Key_Name=>"F1",Command_Name=>"""!Users.John"".Phone(""Bill"")");
```

```
procedure Name (Key_Code : String := "");
```

Displays help for the next key pressed, including a valid key name and any procedures bound to that key.

```
procedure Prompt (Key_Code : String := "");
```

Displays in a Command window the full procedure, with prompts, that is bound to the next key pressed.

This allows you to change the default values for the parameters.

```
procedure Save;
```

Not implemented in the current release of the Environment.

RATIONAL

package Line

```
package Line is
  procedure Beginning_Of     (Offset       : Natural := 0);
  procedure Capitalize       (Repeat       : Integer := 1);
  procedure Center           (Right_Margin : Natural := 0);
  procedure Copy             (Repeat       : Integer := 1);
  procedure Delete           (Repeat       : Integer := 1);
  procedure Delete_Backward  (Repeat       : Integer := 1);
  procedure Delete_Forward   (Repeat       : Integer := 1);
  procedure End_Of           (Offset       : Natural := 0);
  procedure Insert           (Repeat       : Integer := 1);
  procedure Indent           (Repeat       : Integer := 1);
  procedure Join             (Repeat       : Integer := 1);
  procedure Lower_Case       (Repeat       : Integer := 1);
  procedure Open             (Repeat       : Integer := 1);
  procedure Transpose        (Offset       : Integer := 1);
  procedure Upper_Case       (Repeat       : Integer := 1);
  procedure Next             (Repeat       : Integer := 1) renames Cursor.Down;
  procedure Previous         (Repeat       : Integer := 1) renames Cursor.Up;
end Line;
```

Description

Package Line contains operations for manipulating lines in an image. Except for the Delete, Center, and Copy procedures, editing operations affect the line starting at the cursor. The Beginning_Of, End_Of, Delete, Center, and Copy procedures affect the entire line regardless of where the cursor is on the line.

The Transpose procedure affects the current and previous lines without moving the cursor. The Join procedure affects the current line and the next line.

The Indent procedure starts a new line, indented to the previous level of indentation. The Open procedure starts a new line below the cursor but does not affect the cursor's position. The Insert procedure inserts a new line, with the cursor remaining at the first column of the line.

The Beginning_Of and End_Of procedures move the cursor as indicated but use the Offset value to adjust the new cursor position by Offset columns from the beginning or end of a line.

```
procedure Beginning_Of (Offset : Natural := 0);
```

Moves the cursor to the first nonblank character in the line.

This procedure uses the Offset value to adjust the new cursor position by Offset columns from the beginning of the line. When repeated in succession, it toggles the cursor between column 1 and the first nonblank character.

```
procedure Capitalize (Repeat : Integer := 1);
```

Capitalizes each word following the cursor in the next Repeat lines and leaves the cursor at the beginning of the following line.

```
procedure Center (Right_Margin : Natural := 0);
```

Centers from the first nonblank character on the line to the last nonblank character on the line.

This procedure centers the nonblank characters of the entire line between the first column and the specified right margin. If the right margin is specified as 0 (the default), the defined fill column is used (see package Set).

```
procedure Copy (Repeat : Integer := 1);
```

Copies the current line Repeat times and places the new copy immediately below the current line, pushing all remaining lines down in the image.

```
procedure Delete (Repeat : Integer := 1);
```

Deletes the next Repeat lines beginning with the current line. Negative values delete previous lines beginning with the current line.

```
procedure Delete_Backward (Repeat : Integer := 1);
```

Deletes Repeat lines from the cursor back to the left margin and Repeat – 1 full line above the current line.

When Repeat = 1, this procedure deletes characters back to column 1 but not the line itself.

```
procedure Delete_Forward (Repeat : Integer := 1);
```

Deletes from the cursor to the end of the line and Repeat – 1 lines following.

```
procedure End_Of (Offset : Natural := Ø);
```

Moves the cursor after the last nonblank character in the line, if Offset = 0.

This procedure uses the Offset value to adjust the new cursor position by Offset columns from the end of the line toward the beginning of the line.

```
procedure Indent (Repeat : Integer := 1);
```

Opens Repeat new lines before the current line and indents to the previous indentation.

If the cursor is within the current line, the procedure splits the line at the cursor position.

```
procedure Insert (Repeat : Integer := 1);
```

Opens Repeat new lines before the current line.

If the cursor is within the current line, the procedure splits the line at the cursor position. If Repeat = 0, no line is opened. Negative values cause the absolute value of Repeat lines to be before the current line, without splitting the current line at the cursor position.

```
procedure Join (Repeat : Integer := 1);
```

Joins the next Repeat lines to the current line, leaving the cursor on the first character that came from the last line.

```
procedure Lower_Case (Repeat : Integer := 1);
```

Converts the next Repeat lines from uppercase to lowercase starting at the cursor position in the current line and leaves the cursor at the beginning of the following line.

```
procedure Next (Repeat : Integer := 1) renames Cursor.Down;
```

Moves the cursor down Repeat lines.

```
procedure Open (Repeat : Integer := 1);
```

Opens Repeat new lines but does not change the cursor position.

If the cursor is within the line, the line is split. If Repeat = 0, no line is added and the line is not split. If Repeat is a negative value, the line is not split, but the absolute value of Repeat lines are inserted before the current line.

```
procedure Previous (Repeat : Integer := 1) renames Cursor.Up;
```

Moves the cursor up Repeat lines.

```
procedure Transpose (Offset : Integer := 1);
```

Exchanges the current and previous lines.

The cursor position that results depends on the value of the Cursor_Transpose-_Moves session switch. If the value is true, the cursor is moved down one line. If it is false, the cursor position is not changed. See SJM, Session Switches, for more information on session switches. The Offset parameter is currently not implemented.

```
procedure Upper_Case (Repeat : Integer := 1);
```

Converts the next Repeat lines starting at the cursor position in the current line from lowercase to uppercase and leaves the cursor at the beginning of the following line.

RATIONAL

package Macro

```
package Macro is
  procedure Start;
  procedure Finish;
  procedure Execute (Repeat   : Integer := 1;
                     Prior    : Natural := Ø);
  procedure Bind    (Key      : String  := "");
  procedure Save    (Expanded : Boolean := False);
  procedure Restore;
end Macro;
```

Description

A macro is a sequence of editor procedures that can be invoked with a single command. Macros can be bound to a key or referenced as the current macro.

To define a keyboard macro:

1. Execute the Start procedure from a key (by pressing Mark - Begin_Of on the Rational Terminal).
2. Enter a sequence of commands or characters from the keyboard. The editor will remember each keystroke in sequence.
3. Execute the Finish procedure from a key (by pressing Mark - End_Of on the Rational Terminal).

Execute the macro Repeat times with the Execute procedure.

The editor allows only one unbound keyboard macro definition at a time. This macro is sometimes referred to as the ***current macro.*** A new definition replaces an existing definition. However, the current macro can be invoked while a new macro is being defined.

A macro can be bound to a key and stored for the duration of the session with the Bind procedure. Macros can be saved between sessions with the Save procedure. They are written into a file in the user's home world. Typically, the filename identifies the terminal type—for example, Rational_Macros. This file is processed at login.

A macro file can be edited once it has been saved.

Each macro that has been bound has a number. This number is displayed when the Bind procedure is executed. This number is also displayed when the Execute procedure is used to execute macros. The current macro is number 0.

If a macro is executed from an Ada program by calling the Execute procedure, the macro will not be executed until the job completes.

```
procedure Bind (Key : String := "");
```

Binds the current keyboard macro to Key and assigns it a number.

This procedure prompts for a key for binding the macro and displays the number assigned to the macro. The key name is the name of the key for the terminal you are using. You can determine this name by pressing [Help on Key]. The key name will appear in the Message window.

```
procedure Execute (Repeat : Integer := 1;
                   Prior  : Natural := 0);
```

Executes a keyboard macro Repeat times.

If Prior = 0, the current macro is executed. If Prior does not equal 0, the macro with the number indicated by Prior is executed. See the Bind procedure for more information on macro numbers.

If a macro is executed from an Ada program by calling the Execute procedure, the macro will not be executed until the job completes.

```
procedure Finish;
```

Ends a macro definition.

Note that this command must be executed from a bound key and not from a Command window.

```
procedure Restore;
```

Rereads the macro file.

Macro files are read by the system only when the user logs in. Thus, if the user edits the macro file, the new contents of the file will not be known to the system unless it rereads the macro file. The user can execute this procedure to force the system to read the macro file without logging out and logging back in again.

```
procedure Save (Expanded : Boolean := False);
```

Saves the current macro file.

The Expanded parameter saves the macro file in user-readable format, which means that you can edit the macros saved in the file. When you save a macro file, it overwrites the current macro file.

```
procedure Start;
```

Begins a macro definition.

Note that this command must be executed from a bound key and not from a Command window.

---

package Mark

```
package Mark is
  procedure Copy_Top;
  procedure Delete_Top;
  procedure Next          (Repeat : Integer := 1);
  procedure Previous      (Repeat : Integer := 1);
  procedure Push          (Repeat : Integer := 1);
  procedure Rotate        (Repeat : Integer := 1);
  procedure Swap;
  procedure Top;
end Mark;
```

Description

A mark is a remembered image position. The Rational Editor remembers marks as absolute image positions on the quarter plane. Mark positions do not change to adjust for inserted or deleted lines and characters.

Use the Push procedure to set a mark. The Top procedure then returns the cursor to the most recently set mark, returning the image to the screen, if necessary, for the marked image to be visible. The Next and Previous procedures move from mark to mark as they are ordered on the stack. At the bottom of the stack, the Next procedure wraps to the top. At the top, the Previous procedure wraps to the bottom. The Copy_Top procedure makes another copy of the top mark on the stack on the top of the stack. The Delete_Top procedure removes the top mark from the stack. The Swap procedure swaps the top two marks on the stack. The Rotate procedure moves the bottom mark on the stack to the top of the stack.

The mark stack holds up to 100 marks. Since there is only one mark stack for all images, marks can be used to move from image to image as well as from point to point within an image.

```
procedure Copy_Top;
```

Makes another copy of the top mark of the mark stack on the top of the mark stack.

The result is that there are two copies of the top item on the top of the mark stack.

```
procedure Delete_Top;
```

Deletes the top item on the mark stack.

```
procedure Next (Repeat : Integer := 1);
```

Moves to the position specified by Repeat next mark.

At the bottom of the stack, this procedure wraps to the top. If Repeat = 0, it does nothing. If Repeat is a negative value, it moves to the Repeat previous mark.

```
procedure Previous (Repeat : Integer := 1);
```

Moves to the Repeat previous mark.

At the top of the stack, this procedure wraps to the bottom. If Repeat = 0, it does nothing. If Repeat is a negative value, it moves to the Repeat next mark.

```
procedure Push (Repeat : Integer := 1);
```

Sets Repeat marks at the cursor and pushes them onto the stack.

If Repeat = 0 or a negative value, the procedure does nothing.

```
procedure Rotate (Repeat : Integer := 1);
```

Rotates Repeat number of marks from the bottom of the stack to the top of the stack.

If Repeat = 0, the procedure does nothing. If Repeat is a negative value, it rotates Repeat number of items from the top of stack to the bottom of the stack.

```
procedure Swap;
```

Swaps the top two marks on the mark stack.

The mark next to the top of the mark stack becomes the top mark of the mark stack. The top mark of the mark stack becomes the mark next to the top of the mark stack.

```
procedure Top;
```

Moves to the mark at the top of the stack.

RATIONAL

package Region

```
package Region is
  procedure Beginning_Of;
  procedure Capitalize;
  procedure Comment;
  procedure Copy;
  procedure Delete;
  procedure End_Of;
  procedure Fill          (Column  : Natural := 0;
                           Leading : String  := "");
  procedure Finish;
  procedure Justify       (Column  : Natural := 0;
                           Leading : String  := "");
  procedure Lower_Case;
  procedure Move;
  procedure Off;
  procedure On;
  procedure Start;
  procedure Uncomment;
  procedure Upper_Case;
end Region;
```

Description

Package Region contains procedures for making and manipulating selections. To make a selection with the Region package, indicate the beginning with the Start procedure, and then move the cursor to the end and indicate it with the Finish procedure. The commands in package Common.Object (EST) can also be used to make selections. The Rational Editor highlights the selection in a different font. It highlights only one selection at a time. You can use the Off and On procedures to unselect and reselect a selection.

The locations of selections of text can be stored in the hold stack. For further information, see package Hold_Stack.

```
procedure Beginning_Of;
```

Moves the cursor to the beginning of the current selection, if the cursor and the selection are in the same window.

If there is no selection in the current window, the procedure does nothing.

```
procedure Capitalize;
```

Capitalizes the first letter in all the words in the selection.

If the cursor and the selection are not in the same window, the procedure does nothing.

```
procedure Comment;
```

Puts comment characters (--) before the leftmost character of each line in the current selection.

If a line is already commented in that selection, additional comment characters (--) are inserted in front of it.

```
procedure Copy;
```

Copies the current selection to the cursor location.

The selection and the cursor can be in different windows.

```
procedure Delete;
```

Deletes the current selection and pushes it onto the hold stack.

The cursor and the selection must be in the same window.

```
procedure End_Of;
```

Moves the cursor to the end of the current selection.

The cursor and the selection must be in the same window.

```
procedure Fill (Column  : Natural := 0;
                Leading : String  := "");
```

Adjusts the placement of all words in the selection to fill completely the column between the left edge of the image and the defined right margin (the Column parameter).

If Column = 0 (the default), the fill column defined by the Image_Fill_Column session switch is used (described in SJM, Session Switches). The Fill command puts as many words as possible on a line, but it leaves a ragged right margin. It also leaves only one space between words.

The Leading parameter allows a character string to be inserted at the beginning of each line in the selection. This can be used, for example, to insert the comment delimiter, (--), into the selection. If the null string (the default), is used for the parameter, then the leading string defined by the Image_Fill_Prefix session switch is used (described in SJM, Session Switches).

```
procedure Finish;
```

Marks the endpoint of a text selection.

```
procedure Justify (Column  : Natural := 0;
                   Leading : String  := "");
```

Adjusts the placement of all words in the selection to justify the selection flush left between the left edge of the image and the defined right margin (the Column parameter).

If Column = 0 (the default), the fill column defined by the Image_Fill_Column session switch is used (described in SJM, Session Switches). The Justify command puts as many words as possible on a line and inserts spaces so that the right margin is even.

The Leading parameter allows a character string to be inserted at the beginning of each line in the selection. This can be used, for example, to insert the comment delimiter (--) into the selection. If the null string (the default) is used for the parameter, the leading string defined by the Image_Fill_Prefix session switch is used (described in SJM, Session Switches).

By default, the Justify command does not compress extra spaces after the period (.), exclamation mark (!), and question mark (?) when filling an image. However, this default can be changed by modifying the Image_Fill_Extra_Space session switch. (See SJM, Session Switches, for more information on session switches.)

Also by default, the Justify command indents subsequent lines to the indent level of the first line of the region. However, this default can be changed by modifying the Image_Fill_Indent session switch. A value of -1 for this switch (the default) specifies that the indentation of the first line of the region should be used for subsequent lines. Values greater than or equal to 0 indent the region the number of spaces that is the value of the switch.

```
procedure Lower_Case;
```

Converts all characters in the selection to lowercase.

The cursor and the selection must be in the same window.

```
procedure Move;
```

Deletes the current selection and copies it at the cursor.

The cursor and the selection must be in the same window.

```
procedure Off;
```

Unselects the current selection.

The cursor and the selection must be in the same window.

```
procedure On;
```

Reselects a selection that has been unselected with the Off procedure.

The cursor and the selection must be in the same window.

```
procedure Start;
```

Marks the start of a selection.

```
procedure Uncomment;
```

Removes comment characters (--) from lines in which they are the leftmost three nonblank characters.

This procedure does not remove comments from the end of lines containing Ada code. The cursor and the selection must be in the same window.

```
procedure Upper_Case;
```

Converts the selection to uppercase.

The cursor and the selection must be in the same window.

RATIONAL

package Screen

```
package Screen is
   procedure Down          (Repeat   : Integer := 1);
   procedure Left          (Repeat   : Integer := 1);
   procedure Right         (Repeat   : Integer := 1);
   procedure Up            (Repeat   : Integer := 1);
   procedure Dump          (To_File  : String  := ">>NAME<<");
   procedure Redraw;
   procedure Clear;
   procedure Copy_Top;
   procedure Delete_Top;
   procedure Next          (Repeat   : Integer := 1);
   procedure Previous      (Repeat   : Integer := 1);
   procedure Push          (Repeat   : Integer := 1);
   procedure Rotate        (Repeat   : Integer := 1);
   procedure Swap;
   procedure Top;
end Screen;
```

Description

The movement operations of package Screen move the cursor Repeat times in the indicated direction, regardless of window boundaries.

The Clear procedure erases the contents of the screen. The Redraw procedure erases the contents and then repaints the screen.

The Dump procedure captures the contents of the screen in a file specified by the To_File parameter, one screen line per line. Each line contains graphics control sequences embedded in text. The graphics control sequences signify:

- A graphics font for window borders.
- Selected and unselected fonts for plain text, underlined text, and reverse video text.

Package Screen provides a mechanism for saving screens on a stack for retrieval at a later time. The 100 most recent screen saves are kept on the stack.

The screen stack saves the windows currently on the screen, including their size, the cursor location, and the current viewport into the window.

The Push procedure puts the current screen on the stack. The Next and Previous procedures provide a mechanism for moving through the items on the screen stack. They replace the current screen with the Next or Previous screen on the stack. At the bottom of the stack, Next wraps to the top. At the top of the stack, Previous moves to the bottom. The Top procedure retrieves the screen at the top of the stack. The Delete_Top procedure removes the screen at the top of the stack from the stack. The Copy_Top procedure adds another copy of the screen on top of the

stack to the top of the stack. The Rotate procedure moves the screen at the bottom of the stack to the top of the stack. The Swap procedure swaps the top two items on the top of the stack.

If you change the contents of any of the windows, the new contents will be shown when the screen is retrieved from the stack. More specifically, this means that if you abandon a window, it will not be displayed when the screen is retrieved. A blank window will appear in its space. If you have altered the contents of the window—for example, adding text to the window—the new contents of the window will be displayed. The Message window will be displayed in its current state, rather than the state it was in when saved. Also, Command windows will be in the state they were in when last used, rather than the state they were in when the screen was saved.

This feature is useful when you are performing one task with a specific set of screens and then need to perform another task using a different set of screens. You can save and later retrieve either one or both of the sets of screens.

```
procedure Clear;
```

Clears the screen completely and resets the terminal mode to ANSI.

Use this procedure when you need to use the terminal to connect to another system or when modifying terminal characteristics.

```
procedure Copy_Top;
```

Makes another copy of the top item of the screen stack on the top of the screen stack.

The result is that there are two copies of the original top item of the screen stack on the top of the screen stack.

```
procedure Delete_Top;
```

Removes the top item from the screen stack.

The item next to the top of the screen stack becomes the top item of the screen stack.

```
procedure Down (Repeat : Integer := 1);
```

Moves the cursor down Repeat lines, regardless of window boundaries.

When the cursor reaches the last line of the screen, the procedure does nothing.

```
procedure Dump (To_File : String := ">>NAME<<");
```

Copies the contents of the screen to the file specified in the To_File parameter, one screen line per line, as a series of graphics control sequences embedded in text.

The default To_File parameter placeholder must be replaced or an error will result. If the To_File parameter is the null string, the procedure uses the file specified in the Screen_Dump_File switch. If a simple name is not supplied to To_File, the file is created in the user's home world.

```
procedure Left (Repeat : Integer := 1);
```

Moves the cursor Repeat columns to the left, regardless of window boundaries.

When the cursor reaches the left edge of the screen, it stops, regardless of the Repeat value.

```
procedure Next (Repeat : Integer := 1);
```

Retrieves and replaces the current screen with the next Repeat screen from the screen stack.

At the bottom of the stack, this procedure wraps to the top.

```
procedure Previous (Repeat : Integer := 1);
```

Retrieves and replaces the current screen with the previous Repeat screen from the screen stack. At the top of the stack, this procedure wraps to the bottom.

```
procedure Push (Repeat : Integer := 1);
```

Pushes the current screen onto the screen stack Repeat times.

```
procedure Redraw;
```

Clears the screen and then redraws it.

This procedure typically is used following the Clear procedure. The Redraw procedure resets the terminal characteristics to those specified at login.

```
procedure Right (Repeat : Integer := 1);
```

Moves the cursor Repeat columns to the right, regardless of window boundaries.

```
procedure Rotate (Repeat : Integer := 1);
```

Takes Repeat items from the bottom of the screen stack and places them in order on the top.

```
procedure Swap;
```

Swaps the top two items on the screen stack.

```
procedure Top;
```

Retrieves the top item on the screen stack and replaces the current screen with the screen saved on the top of the stack.

```
procedure Up (Repeat : Integer := 1);
```

Moves the cursor up Repeat lines, regardless of window boundaries.

When the cursor reaches the first line of the screen, the procedure does nothing.

---

package Search

```
package Search is
  procedure Previous          (Target      : String  := "";
                               Wildcard    : Boolean := False);
  procedure Next              (Target      : String  := "";
                               Wildcard    : Boolean := False);
  procedure Replace_Previous  (Target      : String  := "";
                               Replacement : String  := "";
                               Repeat      : Integer := 1;
                               Wildcard    : Boolean := False);
  procedure Replace_Next      (Target      : String  := "";
                               Replacement : String  := "";
                               Repeat      : Integer := 1;
                               Wildcard    : Boolean := False);
end Search;
```

Description

Package Search contains both search operations and replace operations. The Next and Previous procedures move through the image in the indicated directions, looking for a string matching the Target string. The Replace_Next and Replace_Previous procedures move similarly through the image with the intent of replacing the Target string with the Replacement string.

All four of these procedures are bound to keys (see the Rational Environment Keymap in Volume 1 of the *Rational Environment Reference Manual*).

The search operations find the next or previous occurrence of the Target string. Using the keys to which these operations are bound, a search for the *n*th occurrence can be accomplished by pressing the key *n* times. The search direction can be reversed without respecifying the Target string by pressing the key that is bound to the search operation of the opposite direction.

The replace operations, when the Repeat parameter is greater than 1, search for and replace Repeat occurrences of the Target string with the Replacement string. If the cursor is currently on (or, if using Replace_Next, one position beyond) an occurrence of the Target string, that occurrence is the first one replaced.

The Target and Replacement strings are entered in the Message window. When one of the keys is pressed, prompts appear in the Message window for the necessary strings. Cursor-movement keys, character- and line-delete keys, and beginning and end keys all work while these search and replace strings are composed. You can also use [Next Item] and [Previous Item] to move between the Target and Replacement strings. The Message window banner also notes that search strings are being composed. The composing of search and replace strings can be interrupted with the [Control][G] key combination. Composing of search and replace strings is completed by pressing the appropriate search or replace key again.

If the search operations are called from Ada programs instead of being initiated from keys, the user will not be prompted for the arguments in the Message window; instead, the arguments supplied in the calls will be used.

The replace operations, when the Repeat parameter equals 1 (the default), search for the first occurrence of the Target string and then stop, awaiting a response. There are five choices at that point:

- The occurrence can be replaced and the next occurrence searched for by pressing one of the replace keys again. The direction of searching can be reversed without respecifying the Target or Replacement strings by pressing the key that is bound to the replace operation of the opposite direction.
- The occurrence can be skipped by pressing one of the search keys. The next occurrence is searched for.
- The occurrence can be replaced and the search discontinued by pressing Numeric 0 followed by the appropriate replace key.
- The occurrence and all remaining occurrences can be replaced by pressing Numeric -, followed by Numeric 1, followed by the appropriate replace key.
- The replace operation can be abandoned completely, bypassing any remaining occurrences of the Target string, by pressing any other key. The command bound to the other key is executed.

The Next and the Replace_Next procedures always leave the cursor one position beyond the Target or Replacement string, respectively. The Previous and Replace_Previous procedures always leave the cursor on the first character of the Target or Replacement string, respectively.

The commands in package Search can use regular expressions. Regular expressions allow pattern matching. A pattern can be specified with special *metacharacters* in the Target string that could match many different strings in the image. Matches can occur within a line; matches never span multiple lines.

The Wildcard parameter specifies whether any metacharacters in either the Target or the Replacement string should be treated as normal characters (false) or as wildcard characters (true).

The metacharacters include:

? Matches any single character.

% Matches any character that is legal in an Ada identifier.

$ Matches any Ada delimiter.

{ Matches beginning of line.

} Matches end of line.

^ Excludes the character or group of characters that follow from the search. For example, the expression ^A means any character except A.

* Matches zero or more occurrences of the previous character or set of characters.

\ Treats the following special character as a regular character.

[] Delimits a class; searches for any of the characters in this group. The class represents one character. For example, [ABC] matches a single character that is A, B, or C. [^ABC] matches a single character that is not A, B, or C. Class searches are case-sensitive.

For any of these procedures, if the Rational Editor cannot find Target, it indicates so in the Message window and leaves the image and cursor position unchanged.

```
procedure Next (Target   : String  := "";
                Wildcard : Boolean := False);
```

Searches forward for the next occurrence of the Target string.

When the Wildcard parameter is false (the default), metacharacters are treated as characters only.

```
procedure Previous (Target   : String  := "";
                    Wildcard : Boolean := False);
```

Searches backward for the most recent occurrence of the Target string.

When the Wildcard parameter is false (the default), metacharacters are treated as characters only.

```
procedure Replace_Next (Target      : String  := "";
                        Replacement : String  := "";
                        Repeat      : Integer := 1;
                        Wildcard    : Boolean := False);
```

Finds the first occurrence of the Target string and awaits a confirm, skip, or abandon request, if Repeat = 1.

If the cursor is already on an occurrence, the occurrence is replaced and the next occurrence searched for. If Repeat is greater than 1, the procedure replaces that many occurrences of the Target string with the Replacement string. When the Wildcard parameter is false (the default), metacharacters are treated as simple characters.

A value of −1 for the Repeat parameter replaces everything from the current cursor location to the end of the image.

```
procedure Replace_Previous (Target      : String  := "";
                            Replacement : String  := "";
                            Repeat      : Integer := 1;
                            Wildcard    : Boolean := False);
```

Finds the previous occurrence of the Target string and awaits a confirm, skip, or abandon request, if Repeat = 1.

If the cursor is already on an occurrence, the occurrence is replaced and the next occurrence searched for. If Repeat is greater than 1, the procedure replaces that many previous occurrences of the Target string with the Replacement string. When the Wildcard parameter is false (the default), metacharacters are treated as simple characters.

A value of −1 for the Repeat parameter replaces everything from the current cursor location to the beginning of the image.

---

package Set

```
package Set is
  procedure Insert_Mode          (On          : Boolean     := True);
  procedure Fill_Mode            (On          : Boolean     := True);
  procedure Fill_Column          (Column      : Positive    := 64);
  procedure Designation_Off;
  procedure Input_From           (File_Name   : String      := "<SELECTION>");
  procedure Input_Logging_To     (File_Name   : String      := ">>Name<<");
  procedure Input_Logging_Off;
  procedure Tab_Off              (Column      : Positive);
  procedure Tab_On               (Column      : Positive);
  procedure Tab_Width            (Size        : Positive    := 4);
  procedure Argument_Prefix;
  procedure Argument_Digit       (Argument    : Integer     := 1);
  procedure Argument_Minus;
end Set;
```

Description

Package Set contains procedures for setting editor parameters. Many of these parameters can also be changed through session switches. See SJM, Session Switches, for more information on session switches.

The Designation_Off procedure converts the indicated (prompt or selection) to plain text.

The Insert_Mode procedure sets insert and overwrite modes. The Fill_Mode procedure, when its parameter specifies true, instructs the editor to wrap text lines whose length exceeds that specified in the Column parameter of the Fill_Column procedure.

The Argument procedures cannot be called from a Command window but must be invoked through key bindings.

The Input procedures all pertain to log files. The Input_Logging_To procedure specifies a File_Name to which the editor should begin recording all user keystrokes. The Input_Logging_Off procedure instructs the editor to stop logging keystrokes. The Input_From procedure instructs the editor to read and execute the keystrokes saved by the Input_Logging_To procedure in File_Name.

If the Input_From procedure is called from an Ada program, the keystrokes in the file execute sequentially when the job performing the call completes.

```
procedure Argument_Digit (Argument : Integer := 1);
```

Specifies a count for the following operation.

The procedure is bound to a key; when entered from a Command window, the procedure has no effect. The count can be used as a repeat count or other parameter to the following operation. Entering a sequence of argument digits produces a number. For example, entering the digits 3 and 5 produces an argument digit of 35.

```
procedure Argument_Minus;
```

Specifies a negative value for the following argument.

The procedure is bound to a key; when entered from a Command window, the procedure has no effect.

```
procedure Argument_Prefix;
```

Indicates to the editor that a command argument is being entered.

The procedure is bound to a key; when entered from a Command window, the procedure has no effect. This procedure takes numbers entered on the main keyboard and converts them to numeric keypad argument digits. For example, this would be used if a keyboard did not contain a numeric keypad or if the numeric keypad keys were bound to something else.

```
procedure Designation_Off;
```

Converts the prompt or selection in which the cursor is located to normal text.

This procedure makes it possible to edit and reuse prompts.

```
procedure Fill_Column (Column : Positive := 64);
```

Sets the desired column width for fill mode or for the Region.Fill and Region.Justify procedures.

The column width can also be specified through a session switch (see SJM, Session Switches).

```
procedure Fill_Mode (On : Boolean := True);
```

Wraps long text lines at the column position specified in the Fill_Column procedure, when true.

The mode is specified only for the current image. The fill mode can also be changed by a session switch (see SJM, Session Switches).

```
procedure Input_From (File_Name : String := "<SELECTION>");
```

Specifies a log file in the File_Name parameter (created with the Input_Logging_To procedure) from which the editor should read and execute keystrokes.

The default is the current selection.

If the Input_From procedure is called from an Ada program, the keystrokes in the file execute sequentially when the job performing the call completes.

```
procedure Input_Logging_Off;
```

Stops logging keystrokes.

```
procedure Input_Logging_To (File_Name : String := ">>Name<<);
```

Logs all keystrokes to File_Name.

The parameter placeholder ">>Name<<" must be replaced or an error will result.

```
procedure Insert_Mode (On : Boolean := True);
```

Sets insert or overwrite modes for the current image.

The insert and overwrite modes can also be changed by session switches (see SJM, Session Switches).

```
procedure Tab_Off (Column : Positive);
```

Removes a tab stop from the specified Column for all images.

The tab stops can also be changed permanently using session switches (see SJM, Session Switches).

```
procedure Tab_On (Column : Positive);
```

Sets a tab stop in the specified Column for all images.

The tab stops can also be changed with session switches (see SJM, Session Switches).

You can see the current tab settings by executing the !Commands.What.Tabs procedure. Tab stops can be added to the settings set by the Tab_Width procedure with the Tab_On procedure.

```
procedure Tab_Width (Size : Positive := 4);
```

Sets the number of spaces between tab stops (in columns) for all images.

The tab width can also be changed (see SJM, Session Switches).

You can see the current tab settings by executing the !Commands.What.Tabs procedure. Tab stops can be added to the settings set by the Tab_Width procedure with the Tab_On procedure.

---

package Window

```
package Window is
  procedure Beginning_Of (Offset   : Natural     := 0);
  procedure Child        (Repeat   : Integer     := 1);
  procedure Copy;
  procedure Delete;
  procedure Demote;
  procedure Directory;
  procedure End_Of       (Offset   : Natural     := 0);
  procedure Expand       (Lines    : Integer     := 4);
  procedure Focus;
  procedure Frames       (Maximum  : Positive);
  procedure Join         (Repeat   : Integer     := 1);
  procedure Next         (Repeat   : Integer     := 1);
  procedure Parent       (Repeat   : Integer     := 1);
  procedure Previous     (Repeat   : Integer     := 1);
  procedure Promote;
  procedure Transpose    (Offset   : Integer     := 1);
end Window;
```

Description

The procedures in package Window provide facilities for managing windows and include a number of commands that control window size and disposition.

A window can be in one of three states (ordered from highest to lowest):

- Frozen (also called *locked*) (banner symbol: @)
- Normal
- Replace (banner symbol: ~)

The Promote and Demote procedures change the state of a window to the next higher or lower level.

In the frozen state, the window will not be replaced and will be split only if there is no other space available for bringing up a new window. In the normal state, the window can become eligible for replacement if it becomes the least recently visited window (in this case, it will automatically be changed to the replace state). If it is in the replace state, the window will be replaced the next time the editor needs to bring a new window on the screen (only one window can be in the replace state).

Four procedures facilitate window management. The Frames procedure sets the number of work windows the editor creates on the screen. The editor treats each frame as a separate area for placing windows. The default configuration is terminal dependent (for the Rational Terminal, it is three frames). The Directory procedure displays a list of active images in the Window Directory. The Window Directory allows various operations on these active images. The Promote procedure elevates

a minor window to its own frame (making it a major window) or changes a major window to the next higher state. The Demote procedure changes a window to the next lower state.

Several operations modify the configuration of windows on the screen. The Join, Transpose, and Delete procedures behave similarly to their counterparts in packages Word and Line.

The Child, Parent, Next, and Previous procedures move between windows. The Beginning_Of and End_Of procedures scroll a window to reposition the cursor to the first or last line of the window.

The editor manages the size, shape, and position of windows. Nevertheless, the editor provides the user a procedure for affecting these window attributes. The Expand procedure increases the size, in lines, of the current or most recently used window. The most recently used window is the one in which a command (other than a cursor-movement command) was last entered.

The Focus procedure restores the screen to the state specified by the current defaults.

Many of the aspects of how windows are displayed on the screen can be tailored with session switches. See SJM, Session Switches, for more information on session switches.

```
procedure Beginning_Of (Offset : Natural := 0);
```

Scrolls the image to move the cursor to the top of the window.

This procedure uses the Offset value to adjust the new cursor position by Offset lines from the top of the window.

```
procedure Child (Repeat : Integer := 1);
```

Moves the cursor Repeat times to the next window in the current frame.

```
procedure Copy;
```

Divides the window into two views on the same image.

The procedure does not duplicate the Command window. The two views are separate frames.

```
procedure Delete;
```

Removes the window in a minor or Command window.

The space for that window is returned to the major window with which it is associated. In a major window, the procedure removes the major window and its associated minor and Command windows from the screen. Windows removed from the screen are not destroyed. Major windows and their contents can be retrieved from the Window Directory or by retrieving their definition. Command windows and their contents can be retrieved by creating the Command window again. Minor windows and their contents can be retrieved by pressing [Definition] from the enclosing Ada unit or library or by using the Window Directory.

```
procedure Demote;
```

Changes the state of the current window to the next lower state.

A window can be in one of three states (ordered from highest to lowest):

- Frozen (banner symbol: @)
- Normal
- Replace (banner symbol: ~)

The Promote and Demote procedures change the state of a window to the next higher or lower level.

In the frozen state, the window will not be replaced and will be split only if there is no other space available for bringing up a new window. In the normal state, the window can become eligible for replacement if it becomes the least recently visited window (in this case, it will automatically be changed to the replace state). If it is in the replace state, the window will be replaced the next time the editor needs to bring a new window on the screen (only one window can be in the replace state).

```
procedure Directory;
```

Displays a list of the currently active images in an image.

This image, called the Window Directory, allows various operations on these active images. See EST, Window Directory, for more information.

```
procedure End_Of (Offset : Natural := Ø);
```

Scrolls the image to move the cursor to the bottom of the window.

This procedure uses the Offset value to adjust the new cursor position by Offset lines from the bottom of the window. If Offset is greater than the number of lines in the window, it moves the cursor to the first line of the file and an error results.

```
procedure Expand (Lines : Integer := 4);
```

Enlarges the window by the specified number of Lines, taking lines from an adjacent frame or neighboring window.

```
procedure Focus;
```

Restores the frame sizes, dividing the screen equally among the current number of frames.

```
procedure Frames (Maximum : Positive);
```

Sets the maximum number of frames the editor will create on the screen.

A session switch controls the number of frames, but this procedure can be used to override that value for the current session.

```
procedure Join (Repeat : Integer := 1);
```

Increases current window size by joining the current and next Repeat frames.

```
procedure Next (Repeat : Integer := 1);
```

Moves the cursor down Repeat frames.

From the bottom frame, the cursor wraps to the top of the screen.

```
procedure Parent (Repeat : Integer := 1);
```

Moves the cursor Repeat times to the previous window in the current frame.

```
procedure Previous (Repeat : Integer := 1);
```

Moves the cursor up Repeat frames.

At the top of the screen, the cursor wraps to the bottom of the screen.

```
procedure Promote;
```

Changes the current window to the next higher state or makes a minor window a major window.

A window can be in one of three states (ordered from highest to lowest):

- Frozen (banner symbol: @)
- Normal
- Replace (banner symbol: ~)

The Promote and Demote procedures change the state of a window to the next higher or lower level.

In the frozen state, the window will not be replaced and will be split only if there is no other space available for bringing up a new window. In the normal state, the window can become eligible for replacement if it becomes the least recently visited window (in this case, it will automatically be changed to the replace state). If it is in the replace state, the window will be replaced the next time the editor needs to bring a new window on the screen (only one window can be in the replace state).

```
procedure Transpose (Offset : Integer := 1);
```

Exchanges the current frame with another.

The default Offset value specifies the previous frame. An Offset value greater than 1 ignores intervening frames. A negative Offset value exchanges the current frame with a next frame.

The cursor position that results depends on the value of the Cursor_Transpose-_Moves session switch. If the value is true, the cursor is left in the window that was the current window or, if no windows are below it, the first window on the screen below the Message window. If the value is false, the cursor is left in the window that replaces the window that was the current window. See SJM, Session Switches, for more information on session switches.

---

package Word

```
package Word is
  procedure Beginning_Of;
  procedure Breaks          (Break_Set       : String  := "";
                             Are_Delimiters  : Boolean := True);
  procedure Capitalize      (Repeat          : Integer := 1);
  procedure End_Of;
  procedure Delete          (Repeat          : Integer := 1);
  procedure Delete_Backward (Repeat          : Integer := 1);
  procedure Delete_Forward  (Repeat          : Integer := 1);
  procedure Lower_Case      (Repeat          : Integer := 1);
  procedure Next            (Repeat          : Integer := 1);
  procedure Previous        (Repeat          : Integer := 1);
  procedure Transpose       (Offset          : Integer := 1);
  procedure Upper_Case      (Repeat          : Integer := 1);
end Word;
```

Description

The operations in package Word manipulate strings of characters separated by word delimiters. The default includes blank and Ada delimiters. Word delimiters can be changed with the Breaks procedure.

Except for the Delete procedure, which erases an entire word, the editing and movement commands affect the next, previous, or current word from the cursor forward to the next end of word or from the cursor back to the previous beginning of word.

```
procedure Beginning_Of;
```

Moves the cursor to the first character of the current word.

```
procedure Breaks (Break_Set       : String  := "";
                  Are_Delimiters  : Boolean := True);
```

Redefines the set of break characters to be used in the current session.

The Break_Set string is an unordered set of characters. Each character, if the Are_Delimiters parameter has been set to true for it, delimits a word boundary. This procedure can be used to tailor the editor to define words for particular uses.

At login, the break characters for the current session are set to the value of the Word_Breaks session switch. By default, the break characters are:

" "#%&'()*+,-./:;<=>?[]_`{|}~"

A value of false for the Are_Delimiters parameter removes the characters specified by Break_Set from the set of break characters for the current session.

```
procedure Capitalize (Repeat : Integer := 1);
```

Capitalizes the first letter of the next Repeat words.

After the operation, the cursor appears on the character after the last target word.

```
procedure Delete (Repeat : Integer := 1);
```

Deletes the next Repeat words.

```
procedure Delete_Backward (Repeat : Integer := 1);
```

Deletes the previous Repeat words from the cursor backward.

```
procedure Delete_Forward (Repeat : Integer := 1);
```

Deletes the next Repeat words from the cursor forward.

```
procedure End_Of;
```

Moves the cursor to the last character of the current word.

```
procedure Lower_Case (Repeat : Integer := 1);
```

Converts the next Repeat words to lowercase.

After the operation, the cursor appears immediately after the last target word.

```
procedure Next (Repeat : Integer := 1);
```

Moves the cursor forward Repeat words.

After the operation, the cursor appears immediately after the Target word.

```
procedure Previous (Repeat : Integer := 1);
```

Moves the cursor back Repeat words.

After the operation, the cursor appears on the first character of the target word.

```
procedure Transpose (Offset : Integer := 1);
```

Exchanges the current and previous words.

The cursor position that results depends on the value of the Cursor_Transpose-_Moves session switch. If the value is true, the cursor is moved to the word delimiter at the end of the exchanged word. If it is false, the position of the cursor is not changed. See SJM, Session Switches, for more information on session switches.

The Offset parameter is currently not implemented and is reserved for future development.

```
procedure Upper_Case (Repeat : Integer := 1);
```

Converts the next Repeat words to uppercase.

After the operation, the cursor appears immediately after the last target word.

```
end Editor;
```

RATIONAL

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

H

J

K

L

M

N

Q

R

S

T

RATIONAL

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

Rational Environment
Reference Manual

Editing Specific Types (EST)

Document Control Number: 8001A-22 (803-002307)

Rev. 1.0, February 1985
Rev. 2.0, December 1985
Rev. 3.0, May 1986
Rev. 4.0, July 1986
Rev. 5.0, 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 Editing Specific Types (EST) book of the *Rational Environment Reference Manual* contains reference information describing some of the commands for editing specific image types provided by the Rational Environment™. It is intended for users who are familiar with the Environment and with Ada® programming. Note that the reference information describing some of the commands for editing all image types are documented in Editing Images (EI) of the *Rational Environment Reference Manual.*

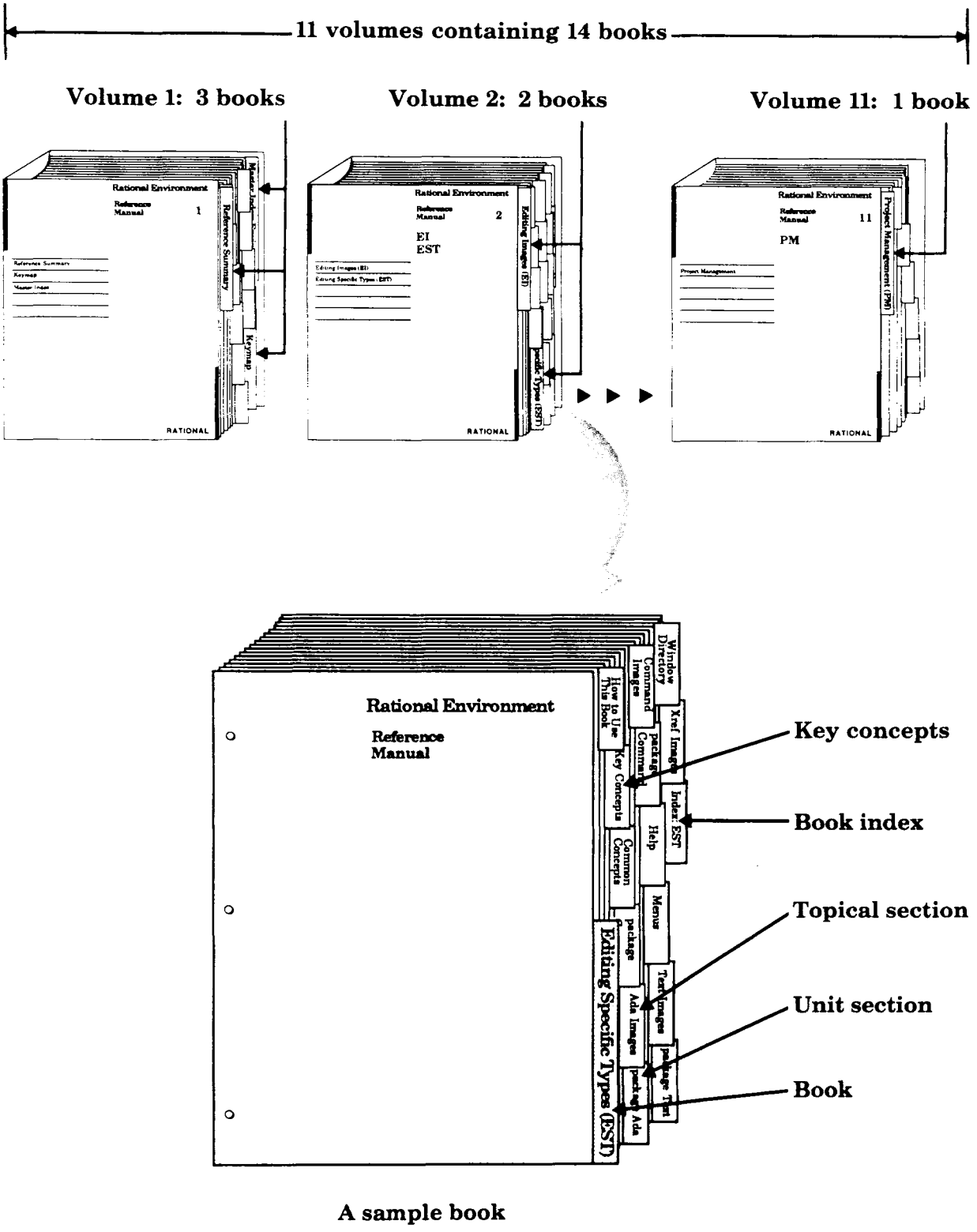
Organization of the Reference Manual

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:

- **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

The title Editing Specific Types refers to using the Rational Editor to edit images based on knowledge about the underlying form and structure of the object or representation for the type of image being edited. This differs from the basic editing operations, available for editing all types of images, that take no advantage of the underlying form or structure of the type. See Editing Images (EI) for more information on these operations.

The Rational Editor knows about many types of images and provides special editing operations for manipulating them. That knowledge is embodied in commands in package Common, provides type-specific editing operations available on most image types, and in other command packages specific to various image types. These commands utilize knowledge about the specific type of image being edited to make the commands more useful for the particular image.

The Rational Editor supports editing of the following images:

- Activity: Lists of subsystem spec and load views.
- Ada: Ada programs or program fragments.
- Command: Commands that are executed in the Rational Environment.
- Debugger: Log of Rational Debugger interactions.
- Help: Help windows that provide information on using the Environment.
- Jobs: The active jobs in the Environment.
- Library: Displays of the libraries in the library system.
- Links: Lists of mappings between Ada simple names and library units in the library system.
- Menu: Lists of Ada declarations.
- Searchlist: Lists of libraries to search for resolving Ada names in commands.
- Switch: Sets of switches that control Environment behavior.
- Text: Text images or files.
- Venture: A collection of work orders intended for a single project or endeavor.
- Windows: The Window Directory list of active images.
- Work list: Lists of work orders.

- Work order: Work orders used in CMVC operations.
- Xref: Lists of Ada compilation units using a particular declaration generated by the !Commands.Ada.Show_Usage procedure.

The common type-specific editing operations provided in package Common are documented in this book. Additional documentation on operations for each specific type of image can be found in books (and in many cases packages) specific to these images. This additional documentation includes a discussion of the type-specific aspects of the common operations from package Common supported for these images.

The !Commands.Activity, !Commands.Work_Order, and !Implementation.Work_Order_Implementation packages are documented in Project Management (PM).

Packages Ada, Command, and Text are documented in this book. In addition, operations on help, menu, window, and xref images are described.

Packages Library and Links, as well as library switches (in package Switches), are documented in Library Management (LM).

Package Search_List and session switches are documented in Session and Job Management (SJM). The What.Jobs and Help displays are documented in package What in SJM and in the Help section in this book.

Package Debug is documented in Debugging (DEB).

For each of these image types, the commands in package Common can have a specific meaning. Each of the packages and sections listed above includes a description of the specific meaning of the commands in package Common that apply to that object. In addition, each command documented in package Common includes a list of the image types for which the command is supported.

Ada Images

This section describes type-specific editing operations for Ada images. Ada images are Ada compilation units or those portions of Ada compilation units that are manipulated using the Environment's incremental compilation operations.

All operations in package !Commands.Common apply to Ada images as well. The procedures from package Common that apply to Ada images are described in this section. In addition to the commands in package Common, commands in package !Commands.Ada apply to Ada images. These commands are also described in this section.

The common editing operations are discussed more fully in the documentation for package Common in this book.

Image Structure

Ada images are composed of Ada elements—the keywords, identifiers, statements, and declarations that make up library units. These elements are built in a treelike hierarchical structure to make up the Ada image. The Rational Editor knows this structure and uses the knowledge to provide specific operations on these images. See the *Reference Manual for the Ada Programming Language* for more information on the syntax and semantics of Ada units.

Key Concepts

Designation

Ada images or their subelements can be designated with specific operations that understand the hierarchical structure of Ada images. Selections are made with the commands in package !Commands.Common.Object or !Commands.Editor.Region. Designations can be made with selections or the cursor position. Sometimes designation through selection is sensitive to whether the cursor is positioned in the selection. See the description of special names to determine which of these options applies to that command.

A unit can also be selected by selecting its stub declaration in its parent (for example, a separate declaration for a subunit in a package body or an entry in a library image for a compilation unit). A unit can also be designated by positioning the cursor on the stub declaration for the unit in its parent.

Selection

Successively larger groups of Ada elements can be selected with the Object.Parent procedure; conversely, successively smaller groups of Ada elements can be selected with the Object.Child procedure. Within a level of selection, specific Ada elements can be selected from the set of elements at that level with the Object.Next and Object.Previous procedures.

As an example, consider the following set of Ada elements (a case statement):

```
case Level is
    when 0 =>
        Emergency_Stop;
    when 1..4 =>
        Need_More;
    when others =>
        null;
end case;
```

Assume that the cursor is on the w in the second when and that no previous selections exist. Executing Object.Parent selects the smallest element enclosing the cursor position; in this case, it is the second arm of the case statement. The section:

```
when 1..4 =>
     Need_More;
```

is highlighted.

Object.Next selects the third arm of the case statement. The third arm is highlighted and the second arm returns to normal video on the screen.

Object.Previous selects the second arm again. It is now highlighted.

Object.Child selects the 1..4 choice of the case statement and moves the cursor to the 1. Object.Child again selects only the 1. Object.Child again leaves nothing selected.

Object.Parent selects the 1 again. Object.Parent a second time selects the 1..4 choice. Object.Parent a third time selects the second arm of the case statement. Executing Object.Parent again highlights all three arms of the example case statement. Executing Object.Parent again selects the entire case statement.

Selections can be used for many operations (for example, to specify the area to be moved or copied). These selection operations also allow traversing the contents of an Ada image.

Cursor Designation

The cursor can be used to designate an Ada element in a way similar to the use of selection. In general, the Rational Editor makes reasonable guesses, based on the cursor location, about the intention of the user. As an example, consider the following Ada element (a subprogram body), assuming there are no selections:

```
with Text_Io;
procedure Test is
Spacing: Text_Io.Positive_Count  := 2
 begin
    Text_Io.New_Line (spacing);
end Test;
```

Assuming that the cursor is anywhere on the line `with Text_Io;`, executing the !Commands.Common.Definition command with the default parameter yields the definition of package Text_Io as if the Ada element `with Text_Io;` were selected.

Similarly, if the cursor is anywhere on the line `Text_Io.New_Line (spacing);` before the period (.), executing the Definition command with the default parameter yields the definition of package Text_Io. However, if the cursor is positioned on the period (.), or anywhere to the right of it on the same line, executing the Definition command brings up the definition of the Text_Io package with the New_Line procedure highlighted as if the Ada element `Text_Io.New_Line (spacing);` were selected.

Special Names

Many of the commands in the Environment use special names. For further information on special names, see Key Concepts.

Unit States

Each Ada compilation unit has a state, called a ***unit state***, associated with it.

A unit can have one of four states in the Environment: archived, source, installed, and coded. These states are ordered; a unit must be promoted from source through installed to coded before it can execute.

The !Commands.Compilation.Demote procedure can be used to place a unit in the archived state, which is much more compact than the source state. To be edited, a unit must be promoted from the archived state to the source state. A unit in the *archived* state has the following significant attributes:

- The unit is not necessarily syntactically correct.
- The unit is not necessarily semantically correct.
- The unit is not known to other units in the system.
- The unit cannot be edited.
- The unit does not have the definition capability and structure-oriented highlighting available to units in the source, installed, and coded states.

When Ada units are created, they are in the source state. When units are demoted because of some dependent unit that needs to be changed, they are also placed in the source state. A unit in the *source* state has the following significant attributes:

- The unit is not necessarily syntactically correct.
- The unit is not necessarily semantically correct.
- The unit can be known to other units in the system.
- The unit can be copied, deleted, moved, or renamed.
- The unit can be changed in any way without affecting any other unit in the system. The only exception is that the name and parameter profile (if the unit is a subprogram) as well as the kind of the unit (for example, subprogram, package, generic) that has its stub installed in its parent cannot be changed.

When an Ada unit is promoted to the installed state, it becomes registered with (or known to) the Environment. The Environment then restricts access to the unit and builds semantic dependencies with other units. A unit in the *installed* state has the following significant attributes:

- The unit is syntactically and semantically correct.
- The unit can be semantically referenced by other units.
- The unit can be copied and it can be deleted, moved, renamed, or demoted if no semantic dependencies are affected.
- The unit can be changed by using incremental addition and change operations (see additional information below).

When an Ada unit is promoted to the coded state, it retains most of the same attributes as the installed state. There is one important addition: the semantically correct unit has machine code generated for it. A unit in the *coded* state has the following significant attributes:

- The unit is known to the Environment.
- The unit is syntactically and semantically correct.
- The unit can be referenced by other units.
- The unit has machine code generated for it by the Environment.
- The unit can be copied and it can be deleted, moved, renamed, or demoted if no semantic dependencies are affected.
- The unit can be changed by using incremental addition and change operations (see additional information below).

unit. Environment.

Commands exist in packages !Commands.Ada and !Commands.Compilation for moving a unit in any state to any other state. The commands in package Ada are for interactive use. They change the state of only one unit at a time. They respond by marking errors on the image of the unit. A unit can be demoted to the archived state only with commands from package Compilation. The commands in package Compilation are for promoting or demoting the state of multiple units, entire libraries, or entire programs. These commands change the state of a unit and any dependent units. They produce error logs and are commonly run as background jobs. Package !Commands.Compilation and further information about unit states and libraries are discussed in Library Management (LM).

Insertion Points

Insertion points are prompts where Ada units can be added to installed or coded Ada units, or they are indications of where elements have been incrementally withdrawn (with demote or edit).

An insertion point has an associated source subunit that is edited to insert new or to contain withdrawn program units, declarations, or statements, or other elements on which incremental compilation is supported (see below). When Ada text associated with an insertion point is promoted higher than the source state, the insertion point is changed into that declaration or statement or other element that corresponds to the element promoted.

Incremental Compilation

The Environment allows certain "compatible" changes to be made to installed and coded units. These changes, referred to as *incremental operations* or *incremental compilation*, allow users to make additions, deletions, and changes to units that have been compiled without requiring that the units being changed be completely recompiled. The incremental compilation facilities supported by the Environment are based on the ability to recompile fragments of Ada units, including statements, declarations, context clause items, comments, and so on. Using these facilities, users can avoid much of the recompilation that would be required if entire compilation units had to be recompiled when any changes were made to them.

Incremental compilation can be used to insert, delete, or demote/edit/repromote (referred to as *withdrawing* and *repromoting*) fragments of Ada units. Insertions are made with insertion points as described above. Deletions are made by selecting the fragment to be deleted and then executing the !Commands.Common.Delete command. Fragments can be withdrawn for editing and repromotion by selecting them and executing the Common.Edit command. This command removes the selected fragment, puts it in a newly created window in the source state for editing and repromotion, and leaves an insertion point in place of the withdrawn fragment in the parent. Once editing is finished in the fragment in the window, promoting the fragment reinserts it in the parent and deletes the window created to hold the fragment.

Incremental Operations on Installed Units

The Rational Environment currently supports the following incremental changes to units in the installed state:

- New declarations that are upwardly compatible (based on Ada semantics) can be inserted. Existing declarations with no dependents can be deleted or demoted from installed to source, edited, and then reinstalled.
- New statements can be inserted in units in the installed state. Existing statements can be deleted or demoted to source, edited, and then reinstalled.
- New context clause items can be inserted if they are upwardly compatible (based on Ada semantics). Existing context clause items with no dependents can be deleted or demoted from installed to source, edited, and then reinstalled or recoded.

- New comments on lines by themselves can be inserted. Existing standalone comments can be deleted or demoted from installed (or coded) to source, edited, and then reinstalled.

Incremental operations are not allowed for two-part types or generic formal parts or generic specifications with installed instantiations. Incremental operations for most declarations are also supported only for manipulations of the entire declaration, not for component parts. Incremental insertion will not work for most pragmas.

Incremental Operations on Coded Units

The Rational Environment supports the following incremental changes to units in the coded state:

- In a library unit specification, new declarations that are upwardly compatible (based on Ada semantics) can be inserted. Also, in a library unit specification, existing declarations with no dependents can be deleted, or they can be edited and reinserted. Because the elaboration code for the declarations in a specification is associated with the corresponding body, incremental insertions or deletions in a library unit specification results in the demotion of the corresponding body to the installed state.
- In a library unit specification, pragmas can be incrementally inserted, deleted, or edited only if all declarations to which the pragma refers are also being simultaneously inserted, deleted, or edited within the same insertion point.
- New context clauses can be inserted if upwardly compatible (based on Ada semantics) only if the units named in the context clause are coded. Existing context clauses with no dependents can be deleted, or they can be edited and then reinserted. Incremental insertion or deletion of context clauses results in the demotion of any dependent main programs.
- Insertion, deletion, and editing of comments is allowed in all coded units.

Note that all restrictions on incremental insertions, deletions, and editing of units in the installed state also apply to units in the coded state.

Versions

Versions of Ada compilation units exist just as versions exist for other objects in the Environment. A new version is created when Edit is pressed for a unit. Versions are not created under any other state change. They are not created when incremental changes are made to a unit.

Committing Images

Not all Ada units need to be committed (saved) explicitly. Ada units that are installed or coded are always permanent; they need not be committed explicitly.

Ada units that are source must be committed to be permanent. Changes made to the unit are made permanent either explicitly when the unit is committed or implicitly when the unit is released or promoted to a higher state.

Locks

The Environment creates locks on Ada units under several circumstances. Whether a lock is created depends on the state of the unit.

If the unit is installed or coded, the unit is never locked. The unlocked unit allows multiple jobs to access the unit. It also allows multiple jobs to make incremental "compatible" changes to the unit.

If the unit is source, the unit can be locked. The two kinds of locks used are read only and write.

With the !Commands.Common.Definition procedure, a read-only lock is used on the unit. This allows multiple jobs to access the unit, but no job can edit the unit.

With the !Commands.Common.Edit procedure, a write lock is used on the unit. This prevents any other job from accessing the unit. Note that a write lock is also placed on a unit in the installed or coded state when the Edit procedure is used, because the Edit procedure demotes the unit to source and then places a write lock on it.

Library Switches

Some of the behavior of the commands for editing Ada images can be tailored with library switches.

The case of identifiers in formatted Ada images is determined by the Keyword_Case library switch. The allowable values for this switch are: Capitalize, Lower, and Upper. The value determines how identifiers are displayed after the !Commands.Common.Format operation.

See LM, package Switches, for more information on library switches.

Commands from Package Common

The following commands from package !Commands.Common are supported for editing Ada images. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Ends the editing of the Ada image. Any changes made to the image since the last commit or promote are lost. However, incremental changes made to installed or coded units, which are permanent as soon as they are promoted, are not lost. The window is removed from the screen and from the Window Directory. The Window parameter specifies the window to be removed from the screen. The default is the current image.

procedure Common.Clear_Underlining

Removes the underlining created by the Common.Semanticize and other procedures. The Semanticize procedure checks the image for semantic consistency, underlining semantic errors.

procedure Common.Commit

Makes permanent any changes to the Ada image. When source Ada images are edited, this procedure saves the changes to the image in the underlying permanent representation. These changes are built in temporary areas until the changes are committed. Then the temporary areas are made a permanent part of the storage hierarchy when a new version of the unit containing the changes is created.

This procedure is used only for Ada images that are in the source state. Direct editing changes (that is, not incremental) to an Ada image in a state other than the source state are not allowed. Changes to the unit caused by promoting or demoting the unit are permanent and need not be committed.

The commit operation is also implicitly performed by the !Commands.Common.Promote, !Commands.Ada.Install_Unit, Ada.Code_Unit (if the unit is source, the operation is not incremental, and the operation completes successfully), and Common.Release procedures.

procedure Common.Complete

Completes the selected Ada identifier or the identifiers in the selected element using Ada's semantics for name resolution. If more than one name can complete the identifier and the Menu parameter is set to true, a list of choices is produced in the menu window. See the description of menus in this book for more information on the editing operations available on menus.

The identifier is completed using the Ada context that exists for the compilation of the selected element, including *with* clauses, *use* clauses, renaming declarations, and so on.

procedure Common.Create_Command

Creates a Command window below the current Ada window if one does not exist; otherwise, it puts the cursor in the existing Command window below the current Ada window. This Command window initially has a *use* clause:

```
use Editor, Ada, Common, Debug;
```

This *use* clause provides direct visibility to the declarations in packages Ada, Common, Debug, and Editor without requiring qualification for names resolved in the command.

procedure Common.Definition

Finds the defining occurrence of the designated element and brings up its image in a window on the screen, typically with the definition of the element selected. If a name is provided to the Name parameter, it is used. If no name is provided, the cursor location is used to designate the element. A read-only lock is acquired on the unit.

The In_Place parameter specifies whether the current frame should be used. The default is false. The Visible parameter specifies whether the specification or body should be displayed. The default, true, specifies that the specification should be preferred.

The procedure finds the most reasonable definition of the element, given the current editing context. If the element is:

- A subunit stub or an insertion point, the corresponding subunit is viewed (if the Visible parameter is false).
- The visible part of a unit, the corresponding body of the unit is viewed (if the Visible parameter is false).
- The body of a unit, the corresponding visible part of the unit is viewed (if the Visible parameter is true).
- A usage of an identifier, the identifier's defining occurrence is viewed (if the Visible parameter is true or false).
- A declaration of an object, the object's type declaration is viewed (if the Visible parameter is true or false).

If the selected or designated item is in a subsystem spec view, then the default session activity is used to find its definition. See Project Management (PM) for more information on subsystems and activities.

The following tables illustrate some additional examples of the use of the Definition command with the Visible parameter true or false. The first column specifies the location of the cursor when the Definition command is executed. The second and third columns specify the effect of setting the Visible parameter to true or false, based on the position of the cursor.

Example 1:

```
package P1 is new P; -- instantiation of generic
     . . .
     P1.B;
```

Table 2-1. Instantiations

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
P1.B (cursor on P1)	instantiation	instantiation
P1.B (cursor on B)	generic specification	generic body

Example 2:

```
type T; -- incomplete type declaration
     . . .
type T is new integer; -- corresponding full type declaration
     X:T;
     . . .
```

Table 2-2. Incomplete Types

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
X:T (cursor on T)	full type declaration	full type declaration
full type declaration	incomplete type declaration	incomplete type declaration
incomplete type declaration	full type declaration	full type declaration

Example 3:

```
type T is private;
     . . .
type T is new integer;
     . . .
     X:T;
```

Table 2-3. Private Types

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
X:T (cursor on T)	private declaration	private part
private part	private declaration	private declaration
private declaration	private part	private part

Example 4:

```
task type T; -- task incomplete type
. . .
task type T is -- specification
. . .
task body T is
. . .
    X:T;
```

Table 2-4. Task Types

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
T1:T (cursor on T)	task specification	task body
entry call	task specification	task body
task specification	task body	task body
task incomplete type	task specification	task body

procedure Common.Demote

Demotes an Ada unit or element to a lower unit state. If there is no selection or if the current selection is for an entire compilation unit, the procedure changes the state of the Ada unit in the current window, assuming there are no dependent units. If there are dependent units, a list of them is displayed in the menu window that is brought onto the screen. See the description of menus in this book 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.

If there is a selection other than the entire unit and if incremental compilation is allowed on the element selected (see the rules on incremental compilation stated above), this procedure removes the element from the parent unit, replaces it with an insertion point, and leaves the element in the source state attached to the insertion point. The source for the element can be visited later by viewing the insertion point.

procedure Common.Edit

Creates a window in which to edit the named or selected Ada unit and demotes the unit to source if necessary.

If there is no selection or if the current selection is for an entire compilation unit or subunit declaration, the procedure creates a window in which to edit the unit, if necessary, and demotes the unit to source if no units depend on the unit. If there are dependent units, a list of them is displayed in the menu window that is brought onto the screen, and the operation fails. See the description of menus in this book for more information on the editing operations available on menus. If the operation succeeds, a write lock is acquired on the unit.

If there is a selection other than the entire unit and if incremental compilation is allowed on the element selected (see the rules on incremental compilation stated above), this procedure removes the element from the parent unit, replaces it with an insertion point, and brings up a new window with the element in it.

The Name parameter specifies the unit to edit. The default is "<IMAGE>". The In_Place parameter specifies whether the current window should be used. The Visible parameter specifies whether the specification or body should be preferred.

procedure Common.Enclosing

Finds the parent or enclosing Ada unit of the current window and displays that parent unit in a window. This procedure acquires a read lock on the unit. The In_Place parameter specifies whether the current window should be used. The Visible parameter specifies whether the specification or body should be preferred. The Library parameter specifies whether the resulting image should be a library.

procedure Common.Explain

Provides an explanation of the error designated by the cursor position in the Ada unit in the current window. Used after syntactic or semantic errors have been discovered, the procedure displays an explanation of those errors in the Message window.

procedure Common.Format

Formats the text in the current window. The procedure redraws some or all of the image after checking for syntactic errors and correcting or prompting for some of the syntactic constructs. If there are syntax errors in the image that cannot be corrected, they are marked as errors.

This procedure adds ending punctuation, including semicolons, right parenthesis, closing quotation marks, *end loop* statements, *end if* statements, and *end* statements for packages and subprograms. Note that *end* statements are usually placed as close to the end of the source as is legal.

Some of the behavior of the commands for editing Ada images can be tailored with session switches.

The case of identifiers in formatted Ada images is determined by the Keyword_Case library switch. Allowable values for this switch are Upper, Lower, and Capitalize.

See package Switches in Library Management (LM) for more information on library switches.

Example 1:

Before the Format procedure:

```
procedure Push is begin
```

After the Format procedure:

```
procedure Push is
begin
    [statement]
end Push;
```

Example 2:

Before the Format procedure:

```
if case when
```

After the Format procedure:

```
if [expression] then
    case [expression] is
        when [expression] =>
            [statement]
    end case;
end if;
```

procedure Common.Insert_File

Copies the contents of the text file specified in the Name parameter into the current Ada image at the current cursor position.

procedure Common.Promote

Promotes the Ada image in the current window to the next higher state. The procedure changes the state of the Ada unit. The specific effect of this procedure depends on the current 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: Execution is attempted if the unit is selected. If parameters are required, the prompt for them appears in the Command window.

If the current window is associated with an insertion point created by incremental compilation and the elements in the window are to be inserted in-line in the parent unit, this procedure causes the elements in the window to be inserted in the parent and the window is deleted.

procedure Common.Release

Ends the editing of the Ada unit. The unit is unlocked, and any changes to the image are committed (made permanent). This window specified by the Window parameter is removed from the screen and from the Window Directory.

procedure Common.Revert

Reverts the Ada image in the current window to the current value of the underlying permanent representation.

procedure Common.Semanticise

Checks the Ada unit for semantic correctness. The procedure checks for compliance with the semantic rules of the Ada language. Errors discovered during semantic checking are underlined.

procedure Common.Object.Child

Selects the Repeat child element of the currently selected element. A child element is one of the elements at the next lower level, in a syntactic sense, from the currently selected element. If an object at that level has not been selected before, the smallest element enclosing the cursor is chosen. If an element at that level has been selected before, the selection is turned off.

procedure Common.Object.Copy

Copies the selected element to the cursor position. The new copy is in the source state. No semantic analysis is done on the selection in its new location, although a check is performed to ensure that a declaration is put in a declarative region and a statement is put in a statement region. Contained units of the copied element are not copied.

procedure Common.Object.Delete

Deletes the selected element. If other elements are dependent on the element because of semantic references (from installed or coded units), the deletion fails and a menu of the dependent units is displayed in the menu window. See the description of the editing operations on menus in this book for more information. 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 currently selected element. The first child is the first one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.

procedure Common.Object.Insert

Creates an insertion point in installed and coded units where statements, declarations, other elements on which incremental compilation operations are supported, or an entire compilation unit can be inserted into the current element.

procedure Common.Object.Last_Child

Selects the last child of the currently selected element. The last child is the last one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.

procedure Common.Object.Move

Moves the selected element to the cursor position. This movement is done by copying the element and then deleting the original element. The new copy is placed in the source state. If other elements are dependent on the element because of semantic references (from installed or coded units), the deletion fails but the copy succeeds. Contained units of the selected unit are not moved.

procedure Common.Object.Next

Selects the Repeat next element past the currently selected element. A next element is the element at the same level, in a syntactic sense, as the current element that appears immediately after the current element. If no such selection can be made, the next element at the enclosing level is selected.

procedure Common.Object.Parent

Selects the parent element of the currently selected element. The parent element is the element that contains the current element at the next higher level, in a syntactic sense, from the current element.

procedure Common.Object.Previous

Selects the Repeat previous element before the currently selected element. A previous object is the object at the same level, in a syntactic sense, as the current element that appears immediately before the current element. If no such selection can be made, the previous element at the enclosing level is selected.

RATIONAL

package Ada

This package contains the set of procedures and types provided for Ada object-specific editing. The commands in package !Commands.Common can also be used for Ada object-specific editing.

procedure Code_Unit

```
procedure Code_Unit;
```

Description

Changes the selected Ada unit or the Ada unit in the current window to the coded state.

The actual effects of this procedure depend on the current state of the unit. If the current state of the unit is:

- Archived: The unit is promoted to the coded state.
- Source: The unit is promoted to the coded state.
- Installed: The unit is promoted to the coded state.
- Coded: The procedure has no effect.

This procedure may involve coding subunits, the parent unit, or the corresponding visible part; however, the transitive closure is not coded.

If the operation succeeds, the unit will be read only.

Units can be demoted to the archived state using the !Commands.Compilation.Demote procedure, with the Goal parameter assigned the value of Compilation.Archived.

Errors

If the unit is in the archived or source state and contains syntactic or semantic errors, the operation will fail.

If another user has a write lock on the unit, the operation will fail.

All referenced specs should be coded to guarantee success.

References

procedure Install_Unit

procedure Source_Unit

LM, procedure Compilation.Demote

LM, procedure Compilation.Make

LM, procedure Compilation.Promote

---

procedure Create_Body

```
procedure Create_Body (Name : String := "<IMAGE>");
```

Description

Inserts a template for the body of the named visible part, the selected visible part, or the visible part in the current window.

This procedure builds a template for the body of the currently selected or named visible part or the visible part in the current window. The template is brought up in a new window, and prompts are provided for statements that must be completed. Any *with* clauses that exist in the visible part are copied into the body template.

Where possible, the body is inserted into the enclosing program unit as an in-line program unit. In-line program units can be changed to separate subunits with the Make_Separate procedure. If the body is built directly into a library, the unit is a compilation unit.

This procedure allows accurate construction of corresponding bodies to existing visible parts. It constructs the skeleton of arbitrary program unit specifications.

Parameters

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

Specifies the visible part for which a body should be built. If no name is provided, the current image is used.

Restrictions

The named unit or current image must be a visible part of a program unit.

Example

Before the Create_Body procedure is executed, the current selection is:

```
function Pop return Boolean;
```

After the Create_Body procedure is executed, a new window is created that contains:

```
function Pop return Boolean is
begin
    [statement]
end Pop;
```

procedure Create_Private

```
procedure Create_Private (Name : String := "<IMAGE>");
```

Description

Inserts a template for the private part of the current package visible part.

This procedure creates the private region of the package. It has no effect if the private part already exists and contains all of the declared private types. The procedure applies recursively to enclosed packages. A prompt is left for the completion of each private type or deferred constant.

Parameters

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

Specifies the name of the package whose private part should be created. The default creates a template for the private part of the current image.

Restrictions

The package must be in the source state.

Example

Before this procedure, a package contained:

```
package A is
    type T is private;
    package B is
        type S is private;
        X : constant S;
    end B;
end A;
```

If the private part of either type has not been completed, possibly because semantic analysis found the above package to be incomplete, the Create_Private procedure is executed. The package now contains:

```
package A is
    type T is private;
    package B is
        type S is private;
        X : constant S;
    private
        type S is new [expression];
    X : constant S := [expression];
    end B;
private
    type T is new [expression];
end A;
```

procedure Delete_Blank_Line

```
procedure Delete_Blank_Line (Repeat : Positive := 1);
```

Description

Not currently implemented.

Specifies that Repeat blank lines should be deleted after the current line (that is, the line on which the cursor is currently located). If one of the specified lines is a nonblank line, the deletion stops, and the nonblank line is not deleted.

procedure Diana_Edit

```
procedure Diana_Edit (Name : String := "<CURSOR>");
```

Description

For Rational internal use.

Shows a read-only image of the internal form of the DIANA tree corresponding to the node given, based, by default, on the cursor position. This can be used only on Ada units or in Command windows.

Parameters

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

Specifies the name of the image. The default, `"<CURSOR>"`, specifies the node on which the cursor is currently located.

procedure Get_Errors

```
procedure Get_Errors;
```

Description

Causes the error underlines in the image to be redisplayed on the image.

These error underlines result from errors in the image. To remove error underlines, use the Common.Clear_Underlining procedure.

This procedure is useful to display underlining resulting from errors after the Clear-_Underlining procedure has been used.

This command is very useful in the following scenario. Assume that you have a unit that has no syntactic errors and many semantic errors, so you make changes to the unit to correct semantic errors. Inadvertently, you add a syntactic error, which is underlined when you press [Format]. Because all the semantic errors are still underlined, you may find it difficult to locate the syntactic error. Thus, you may want to execute the Common.Clear_Underlining command to remove all underlines. Then you can press [Format], which displays the underlining for the syntactic error. Once you have corrected the syntactic error, you can press [Format] again to remove the underlining for syntactic errors that have been corrected. Then you can use the Get_Errors procedure to display the semantic errors again so that you can continue correcting them.

Restrictions

This procedure displays only the latest errors found by the Common.Semanticize procedure.

References

procedure Common.Clear_Underlining

procedure Insert_Blank_Line

```
procedure Insert_Blank_Line (Repeat : Positive := 1);
```

Description

Not currently implemented.

Specifies that Repeat blank lines should be inserted before the current line (that is, the line on which the cursor is currently located). When the insertion has completed, the cursor will be on the first blank line inserted.

procedure Install_Stub

```
procedure Install_Stub;
```

Description

Installs the declaration or stub in the parent unit or library for the unit in the current window.

You might want to install the stub for a unit so that it appears in the library structure or to create a link for it. For subunits, you might want to install the stub so the subunit can be compiled against.

If the unit cannot be installed but the stub can be, this procedure installs the stub (makes the insertion point into a declaration for the unit) without additional semantic checking of the unit.

References

procedure Common.Object.Insert

procedure Install_Unit

```
procedure Install_Unit;
```

Description

Changes the selected or designated Ada unit, or the Ada unit in the current window, to the installed state.

The actual effect of this procedure depends on the current state of the Ada unit. If the unit is in the following state:

- Archived: The unit is promoted to the installed state.
- Source: The unit is promoted to the installed state.
- Installed: The procedure has no effect.
- Coded: The unit is demoted to the installed state.

If other units depend on the unit and the unit is to be demoted, the operation fails and the dependent units are displayed in the menu window that is brought onto the screen. See the description of menus in this book for more information on the editing operations available on menus.

Units can be demoted to the archived state using the !Commands.Compilation.Demote procedure, with the Goal parameter assigned the value of Compilation.Archived.

Errors

If the unit is in the archived or source state and contains syntactic or semantic errors, the operation will fail.

If all referenced specs are not installed or coded, the operation will fail.

References

procedure Code_Unit

procedure Install_Stub

procedure Source_Unit

LM, procedure Compilation.Demote

LM, procedure Compilation.Promote

---

procedure Make_Inline

```
procedure Make_Inline;
```

Description

Changes the subunit in the image or selected subunit stub from a subunit to an in-line program unit.

This procedure removes the *separate* clause and moves the body for the subunit into the parent unit. The library entry for the subunit is removed from the library structure.

Program units can be created as either in-line program units or separate subunits. The Make_Inline procedure allows a program unit that is a separate subunit to be changed to an in-line program unit.

Restrictions

The unit must be in the source or installed state.

Example

If an Ada unit contains the following subunit stub:

```
package body Test is
    ...
    procedure Input_File is separate;
    ...
```

and that subunit has the following contents:

```
separate (Test);
procedure Input_File is
begin
    File_Io.Input;
end Input_File;
```

then placing the cursor in a window that contains the subunit, or selecting the subunit stub and then executing the Make_Inline procedure, removes the subunit stub and replaces it with the contents of the program unit as follows:

```
package body Test is
    ...
    procedure Input_File is
    begin
        File_Io.Input;
    end Input_File;
    ...
```

procedure Make_Separate

```
procedure Make_Separate;
```

Description

Changes the selected subprogram from an in-line program unit to a separate subunit.

This procedure replaces the body with a ***separate*** clause and creates a separate library entry containing the body for the unit.

Program units can be created as either in-line program units or separate subunits. The Make_Separate procedure allows a program unit that is an in-line program unit to be changed to a separate subunit.

Restrictions

The unit must be in the source or installed state.

Example

If an Ada unit in the source state contains the following in-line program unit:

```
package body Test is
    ...
    procedure Input_File is
    begin
        File_Io.Input;
    end Input_File;
    ...
```

then selecting the entire in-line procedure unit (the entire Input_File procedure) and executing the Make_Separate procedure creates a subunit stub as follows:

```
package body Test is
    ...
    procedure Input_File is separate;
    ...
```

and creates the subunit associated with the stub as follows:

```
separate (Test);
procedure Input_File is
begin
    File_Io.Input;
end Input_File;
```

procedure Other_Part

```
procedure Other_Part (Name     : String  := "<IMAGE>";
                      In_Place : Boolean := False);
```

Description

Finds and displays the other part of the named program unit or the program unit in the current window.

This procedure finds the corresponding visible part if the named program unit is a body, or finds the corresponding body if the named program unit is a visible part, and displays this other part in a window. If no program unit is named, the program unit in the current window is used. If no program unit is named and the current window does not contain a program unit, then the procedure has no effect.

Parameters

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

Specifies the program unit whose other part is desired. The default is to use the program unit in the current window.

`In_Place : Boolean := False;`

Specifies whether the current frame should be used to bring up the image. The default specifies that the least recently used frame should be used.

procedure Replace_Id

```
procedure Replace_Id (Old_Id : String := ">>OLD NAME<<";
                      New_Id : String := ">>NEW NAME<<");
```

Description

Replaces all identifiers that match the first string with the second string in the current selection.

This procedure is similar to, but more selective than, a general replace mechanism. It changes Ada identifiers (not comments or identifier fragments) in the current window.

Parameters

`Old_Id : String := ">>OLD NAME<<";`

Specifies the identifier to be replaced. It is not case-sensitive. The default parameter placeholder ">>OLD NAME<<" must be replaced or an error will result.

`New_Id : String := ">>NEW NAME<<";`

Specifies the new identifier. The default parameter placeholder ">>NEW NAME<<" must be replaced or an error will result.

Restrictions

The replacement is confined to the highlighted area.

This command requires the unit to be in the source state.

The user must be able to acquire a write lock on the unit for the operation to complete successfully.

Old_Id must be a simple identifier. New_Id can be any name—an identifier, a qualified name, command reference, and so on.

Example

Given the following procedure:

```
procedure Proc is
begin
    Dure;
end Proc;
```

the Replace_Id procedure can be used to replace both the name of the procedure and the name of the procedure called by selecting the procedure and executing:

```
Replace_Id ("proc", "foo");
Replace_Id ("dure", "bar");
```

The procedure is as follows:

```
procedure Foo is
begin
    Bar;
end Foo;
```

Compare the behavior of this procedure with the action of a typical string replacement capability. After a string replacement of the same strings in the example procedure, the procedure might look like this:

```
fooebar foo is
begin
    bar;
end foo;
```

---

procedure Show_Usage

```
procedure Show_Usage (Name    : String  := "<CURSOR>";
                      Global  : Boolean := True;
                      Limit   : String  := "<ALL_WORLDS>";
                      Closure : Boolean := False);
```

Description

Determines the actual usages, within the scope defined by Limit, of the declaration specified in the Name parameter and displays a list of these units in an xref window.

The usages can be seen by taking the definition of the entries in the xref, which brings up images of these units in windows, with each actual usage of the declaration underlined. The !Commands.Editor.Cursor.Next and Editor.Cursor.Previous procedures can be used to move between these usages. For more information, see the description of the editing commands available on xrefs in Xref Images in this book.

Note that if the actual usages are only in the current unit, each usage is underlined and an xref window is not created.

Parameters

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

Defines the identifier whose usages are to be displayed. By default, the Show_Usage procedure determines the usages of the designated identifiers in the current unit and in all units containing compiled references to the defining occurrence of the identifier.

```
Global : Boolean := True;
```

Specifies, when true, that units other than the unit specified should be marked. When false, this parameter specifies that only the unit named should be marked.

```
Limit : String := "<ALL_WORLDS>";
```

Specifies, when the Global parameter is true, the limit for the range of units. The default, "<ALL_WORLDS>", specifies that references in any world should be included. Other special names that can be used for this parameter are:

"<UNITS>"	Specifies only the units named in the operation.
"<SUBUNITS>"	Specifies only the units named in the operation and their subunits.
"<DIRECTORIES>"	Specifies only the units in the same set of directories as the units specified to the operation.
"<WORLDS>"	Specifies only the units in the same world as the units specified to the operation.
"<ALL_WORLDS>"	Specifies a unit in any world.

```
Closure : Boolean := False;
```

Specifies whether the command should show indirect references to a unit. For type declarations, it shows derived types, subtypes, and accesses to type. For packages or subprograms, it shows renames. The default, false, specifies that indirect references should not be shown.

Restrictions

This procedure finds usages only in installed or coded units.

---

procedure Show_Unused

```
procedure Show_Unused (In_Unit           : String  := "<IMAGE>";
                       Check_Other_Units : Boolean := True);
```

Description

Shows the declarations that are not referenced in a unit.

The unused references are underlined.

Parameters

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

Specifies the unit to check for unused references. The default is the current image.

`Check_Other_Units : Boolean := True;`

Specifies whether to check other units that directly or indirectly ***with*** In_Unit for usage of the declarations. The default, true, specifies that other units should be checked.

Restrictions

The unit must be in the installed or coded state.

The referencing units must be in the installed or coded state.

procedure Source_Unit

```
procedure Source_Unit;
```

Description

Changes the Ada unit in the current window, or the selected or designated unit, to the source state.

The actual effects of this procedure depend on the current state of the unit. If the current state of the unit is:

- Archived: The unit is promoted to source.
- Source: The procedure has no effect.
- Installed: The unit is demoted to the source state.
- Coded: The unit is demoted to the source state.

If other units depend on the unit and the unit is to be demoted, the operation fails and the dependent units are displayed in the menu window that is brought onto the screen. See the description of menus in this book for more information on the editing operations available on menus.

Units can be demoted to the archived state using the !Commands.Compilation.Demote procedure, with the Goal parameter assigned the value of Compilation.Archived.

This procedure brings a new unit to the source state and places an entry for it in the library structure under its proper name. The write lock on the unit is released (the unit is read only).

References

procedure Code_Unit

procedure Install_Unit

LM, procedure Compilation.Demote

LM, procedure Compilation.Make

LM, procedure Compilation.Promote

procedure Withdraw

```
procedure Withdraw (Name : String := "<IMAGE>");
```

Description

Withdraws the stub (library entry for) of the named Ada unit, the selected Ada unit from its parent, or the Ada unit in the current window and demotes the unit to the source state.

The stub is replaced with an insertion point.

The actual effects of this procedure depend on the current state of the unit. If the current state of the object is:

- Archived: The procedure has no effect.
- Source: The stub declaration is withdrawn from the enclosing unit or library.
- Installed: The unit is demoted to the source state and the stub declaration is withdrawn from the enclosing unit or library.
- Coded: The object is demoted to the source state and the stub declaration is withdrawn from the enclosing unit or library.

If other units depend on the unit and the unit is to be demoted, the operation fails and the dependent units are displayed in the menu window that is brought onto the screen. See the description of menus in this book for more information on the editing operations available on menus.

The user may need to withdraw a unit to change its name or its parameter profile.

Parameters

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

Specifies the name of the unit to be withdrawn. The default is the current image.

Restrictions

A subunit cannot be withdrawn to be made a child of another unit; however, it can be withdrawn for a change to be made to its name.

Example 1

The Withdraw procedure can be useful in changing the name of a unit.

To rename a unit:

1. Use the Withdraw procedure to withdraw the unit.
2. Edit the name as appropriate.
3. Repromote the unit to the desired state.

Example 2

The Withdraw procedure can be used to change the name of both the spec and the body of a unit or their parameter profiles.

To rename both parts:

1. Use the Withdraw procedure to withdraw the body.
2. Use the Withdraw procedure to withdraw the spec.
3. Edit the spec as appropriate.
4. Promote the spec to the desired state.
5. Edit the body as appropriate.
6. Promote the body to the desired state.

---

end Ada;

Command Images

This section describes type-specific editing operations for command images in Command windows. Structurally similar to Ada units, commands are used to run Ada programs (which can be system commands or user programs). Commands do not have permanent underlying representations in the directory system (although historical records are maintained for perusal and editing/reexecution; see below).

Commands are created in a Command window with the Common.Create_Command procedure. You can edit them with with common editing operations from package !Commands.Common. The common commands that apply to commands are documented in this section. Additional editing operations available for commands in package !Commands.Command are documented after this section.

Image Structure

Commands consist of Ada block statements with special treatment of the declarative/exception parts and sequence of statements. Here is a typical example of a command from a Command window created from an Ada unit:

```
declare
    use Editor, Common, Ada, Debug;
begin
    [Ada-statement]
end;
```

By default, the declarative region contains a *use* clause for packages !Commands.Editor and !Commands.Common. Based on the type of image the Command window is associated with, more entries can be added automatically to this *use* clause to give direct visibility to additional type-specific editing operations. In the above example, the commands in package !Commands.Ada and !Commands.Debug are also visible because of the addition of Ada and Debug to the *use* clause.

The statement prompt indicates where Ada statements can be entered. These statements are compiled and executed when the command is committed or promoted by pressing Commit or Promote (this is called *executing a command*).

All statements entered in the statement region of the block become prompts when the command is executed. This allows new commands to be entered over the old

commands in the prompt and facilitates reexecuting the existing command or turning it into text, editing it, and then executing it.

Changes can also be made to the declarative region of the block. These changes do not become prompts when the command is executed, and they stay for the life of the window.

Executing Command Windows

After Promote or Commit is pressed, the Command window, actually an Ada declare block, is semanticized. First, declarations are checked to determine if they are local declarations. If the declarations are not local, simple names (for example, Access_List) are resolved using the searchlist. String literals used as a name prefix (for example, "!Commands.Access_List".Display) are resolved in the current context. (Note that string literals used as a name prefix must be the name of a library.) The current context is the closest enclosing library for the image in the major window to which the Command window is attached. If the image in the major window is a library, that library is the current context, rather than the enclosing library.

From an I/O window, the context for the Command window is that of the last job that read from or wrote to that window.

Key Concepts

Designation

Commands have the same form and structure as Ada images. Designation for commands is identical with that for Ada (see Ada Images, in this book).

Unit States

Commands do not have unit states as Ada units do. However, the action of committing or promoting a command implicitly promotes it from source to coded, elaborates and executes it, and then demotes it to source when execution terminates.

Versions

Commands do not have versions because there is no underlying permanent object.

Histories

Previous commands are remembered for the life of a Command window. This history of commands allows them to be reviewed, edited if necessary, and then reexecuted. Once a command is executed in a Command window, it is remembered in the history and the statements in the block are displayed as a prompt. The command can be reexecuted with the prompt value for the statements or the prompt can be changed to a new value. These changes are recorded in the history when the command is next executed.

To return to the `[statement]` prompt, press Edit. The command currently displayed in the Command window will be replaced with the `[statement]` prompt.

The history is maintained as a series of commands. The !Commands.Common.Undo procedure steps backward in the series; the !Commands.Common.Redo procedure steps forward in the series. At each step, the command can be executed or used in constructing a new command.

Library Switches

Some of the behavior of the commands for editing commands can be tailored with library switches.

The case of identifiers in formatted commands is determined by the Keyword_Case library switch. Allowable values for this switch are Upper, Lower, and Capitalize.

See LM, package Switches, for more information on library switches.

Commands from Package Common

The following commands from package !Commands.Common are supported for editing commands. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Ends the editing of the command and removes the Command window from the screen. The Window parameter specifies which window should be removed from the screen. The default is the current image, which is the window above which or on which the command is executed.

procedure Common.Clear_Underlining

Removes all underlines in the current Command window.

procedure Common.Commit

Executes the command in the Command window by formatting, semanticizing, and coding it. This procedure is identical to the Promote procedure.

procedure Common.Complete

Completes the Ada fragment designated by the cursor using Ada semantics for name resolution. If more than one name could complete the identifier, and the Menu parameter is true, a list of choices is produced in the menu window. See the description of menus in this book for more information on the editing operations available on menus.

This procedure completes only names of identifiers that are declared in the fragment in the Command window or that are visible through the searchlist. See package Search_List in Session and Job Management (SJM) for more information on searchlists.

Declarations in Command windows must be selected before the Complete command will actually complete them. Statements need not be selected for the Complete command to complete them.

procedure Common.Create_Command

Creates a Command window below the current Command window if one does not exist; otherwise, it puts the cursor in the existing Command window below the current Command window. This Command window initially has a *use* clause:

```
use Editor, Command, Common;
```

This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, !Commands.Common, and !Commands.Command without requiring qualification for names resolved in the command.

For certain other window types, other packages are added to the *use* clause to provide visibility to operations used to edit those kinds of windows.

procedure Common.Definition

Finds the defining occurrence of the named or designated element and brings up its image in a window on the screen, typically with the definition of the element selected. The Semanticize or Promote procedure must have been executed on the window containing the named or designated element. If a name is provided for the Name parameter, it is used. If no name is provided, a selection is used if one exists. Otherwise, the cursor location is used to designate the element.

The procedure finds the most reasonable definition of the element, given the current editing context and the value of the Visible parameter. The Name parameter specifies which element's definition should be given. The In_Place parameter specifies whether the current window should be used, and the Visible parameter specifies whether the specification or the body should be preferred.

procedure Common.Demote

Ends the editing of the command. The contents of the Command window are destroyed and the original contents are restored. All history is lost.

procedure Common.Edit

Replaces the contents of the Command window with a `[statement]` prompt.

procedure Common.Enclosing

Finds the major window to which the Command window is attached and puts the cursor in it. The In_Place parameter specifies whether the current window should be used. The Library parameter specifies whether the enclosing object should be a library.

procedure Common.Explain

Provides an explanation of errors in the command in the current window. Used after syntactic or semantic errors have been discovered, the procedure displays an explanation of those errors in the Message window.

procedure Common.Format

Formats the text in the current Command window. The procedure redraws some or all of the image after checking for syntactic errors and correcting or prompting for some of the syntactic constructs. If there are syntactic errors in the image that cannot be corrected, they are marked as errors.

Some of the behavior of the commands for editing commands can be tailored with session switches.

The case of identifiers in formatted Ada images is determined by the Keyword_Case library switch. Allowable values for this switch are Upper, Lower, and Capitalize. Only one of these switches should have the value of true. The switch that is true determines how identifiers are displayed after the Format operation.

See package Switches in Library Management (LM) for more information on session switches.

Example 1:

Before the Format procedure:

```
declare
    use Editor, Common, Ada;
begin
    if
end;
```

After the Format procedure:

```
declare
    use editor, Common, Ada;
begin
    if [expression] then
        [statement]
    end if;
end;
```

procedure Common.Insert_File

Copies the contents of the named text file into the Command window at the current cursor position.

procedure Common.Promote

Executes the command by formatting, semanticizing, and coding it. This command is the same as the Commit procedure.

procedure Common.Redo

Recalls commands entered in a Command window after the Undo procedure is executed on the Command window. The Rational Editor remembers changes made to command images since the Command window was created. Each execution of the Complete, Edit, Demote, and Revert procedures marks another change, as well as execution of the command. The Repeat parameter specifies the number of commands to move forward in the history.

procedure Common.Release

Ends the editing of the command. The Command window is destroyed and removed from the screen. All history is lost. The Window parameter specifies the window to be released.

procedure Common.Revert

Redraws the command in the current window.

procedure Common.Semanticize

Checks the command for semantic correctness. The procedure checks for compliance with the semantic rules of the Ada language. Errors discovered during semantic checking are underlined. This procedure must be executed either directly or indirectly (by using the Promote procedure) before the Definition procedure will execute successfully.

procedure Common.Undo

Recalls changes previously made to a command. The Rational Editor remembers changes made to command images since the Command window was created, called a history. Each execution of the Edit, Demote, and Revert procedures marks another change, as well as each execution of the command. As commands are undone, the last undone command in the history becomes the place where new user-entered commands are saved in the history. These changes can be reinstated with the Redo procedure. The Repeat parameter specifies the number of commands to move backward in the history.

procedure Common.Write_File

Copies the contents of the selection in the Command window to the file specified by the Name parameter.

procedure Common.Object.Child

Selects a child element of the currently selected element. A child element is one of the images at the next lower level, in a syntactic sense, from the current element. If an element at that level has not been selected before, the element on which the cursor is currently located is chosen. If an element at that level has been selected before, the selection is turned off.

procedure Common.Object.First_Child

Selects the first child of the currently selected element. The first child is the first one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.

procedure Common.Object.Last_Child

Selects the last child of the currently selected element. The last child is the last one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.

procedure Common.Object.Next

Selects the Repeat next element past the currently selected element. A next element is the element at the same level, in a syntactic sense, as the current element that appears immediately after the current element. If no such selection can be made, the next element at the enclosing level is selected.

procedure Common.Object.Parent

Selects the parent element of the currently selected element. The parent element is the element that contains the current element at the next higher level, in a syntactic sense, from the current element.

procedure Common.Object.Previous

Selects the Repeat previous element before the currently selected element. A previous element is the element at the same level, in a syntactic sense, as the current element that appears immediately before the current element. If no such selection can be made, the previous element at the enclosing level is selected.

RATIONAL

package Command

This package contains the set of procedures and types provided for command object-specific editing. The commands in package !Commands.Common can also be used for command object-specific editing.

procedure Debug

```
procedure Debug;
```

Description

Executes the program named in the Command window or the selected program in a library with debugging enabled.

This procedure is normally bound to the [Meta][Promote] key combination on the Rational Terminal keyboard, which executes the program with the Rational Debugger enabled (see the Keymap in Volume 1 of the *Rational Environment Reference Manual*).

If a selection is used to specify the program to execute, the program must not require the user to enter parameter values. If the program does require entry of parameters, an error message will result.

If the Rational Debugger is currently connected to a job, that job is killed. This behavior is controlled by the Debug_Kill_Old_Jobs debugger flag (see Debugging (DEB) for more information). If the Debug_Kill_Old_Jobs flag is true, the default value, then the job being debugged is killed when a new Debug call is made. If false, the current job is released and continues its normal execution.

If the Debug_Require_Debug_Off debugger flag is true and if a job is being debugged, this command fails. With this flag true, the job currently being debugged must first be eliminated with the !Commands.Debug.Debug_Off procedure. For further information on this procedure, see DEB, package Debug.

As part of executing the Debug procedure:

- The control and evaluation contexts are cleared.
- Any stepping operations in progress for the old job are canceled.
- Tracing requests are canceled.
- Exception-handling requests are cleared (controlled by the Save_Exceptions debugger flag).
- The default, catch all exceptions request, is installed.
- Any active breakpoints are deactivated.

This procedure differs from the !Tools.Debug_Tools.Debug_On procedure, which enables debugging only if the task calling Debug_On is part of the job being debugged and never results in a job being killed. For further information on the Debug_On procedure, see DEB, package Debug_Tools.

Example

To debug a program started by the Io_Simulator command, enter in a Command window:

```
Io_Simulator;
```

and then either create a new Command window attached to the first and execute the Debug procedure or press the key to which this procedure is bound (normally the Meta Promote combination).

You can now debug the program using debugging commands and the Debugger window.

References

DEB, procedure Debug.Debug_Off

DEB, type Debug.Option, enumeration Kill_Old_Jobs (see also procedure Debug.Enable)

DEB, type Debug.Option, enumeration Require_Debug_Off (see also procedure Debug.Enable)

DEB, type Debug.Option, enumeration Save_Exceptions (see also procedure Debug.Enable)

DEB, procedure Debug_Tools.Debug_On

procedure Spawn

```
procedure Spawn;
```

Description

Spawns a background job to execute the command in the Command window or a program selected in a library window.

This procedure is the same as the !Commands.Common.Promote procedure for Command windows, except that the Spawn procedure executes the Command window as a background job.

If a selection is used to specify the program to execute, the program must not require the user to enter parameter values. However, it will accept parameters with default values, including special names. If the program does require entry of parameters, an error message will result.

This procedure is similar to executing a Command window normally (with the Common.Promote procedure) and then interrupting that execution with the !Commands.Job.Interrupt procedure. The job is run as a background job, and control of the cursor is returned to the user immediately.

end Command;

Common Concepts and Operations

This section describes the commands and key concepts for type-specific editing commonly available for images.

Several concepts apply to all type-specific editing operations in the Environment. These concepts pertain to commands executed from the Rational Editor as well as to the specific objects or items being edited.

For many of these concepts, the exact implementation may differ slightly for each particular image type. Examples of these concepts, details for each type, and the editing commands available for each type are contained in the documentation for the specific image types in this book.

Image Types

The image type is displayed in the banner for the window, just to the right of the name enclosed in parentheses. This type determines the type-specific editing operations available on the image in the window. Because Command windows have no banners, their image type is implicit and is not displayed.

Command windows opened from these images have *use* clauses that provide direct visibility to the commands supported on a type-specific basis. *Use* clauses for packages !Commands.Editor and !Commands.Common are always present. Additional *use* clauses will be added based on the specific type of image with which the Command window is associated.

Designation

Because the Rational Editor knows the form and structure of different types of images, it provides operations for designating structural components of the image. The two ways of designating structural components are:

- Selection performed with the commands in package Common.Object or package Editor.Region (for further information, see EI, package Editor).
- Cursor position.

Designation is used by several commands as an alternative to naming an item. When an item is being moved or copied, designation can be used instead of naming the object explicitly.

Special Names

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

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

Versions

The Environment maintains multiple versions for each object in the directory system. Typically a new version of an Ada unit is created each time the image is first edited (see "Committing Images," below). New versions of text files are created when the file is committed or promoted. Version numbers are assigned by the Environment when each version is created.

One version of each object is called the *default version*. This is typically the newest version, but it can be any version. When an object is edited or viewed, the default version of the object is used. All other versions of the object are considered deleted. Any particular version of an object can be undeleted, in which case it becomes the default version and the previous default version is deleted.

The Environment allows a specified number of old, deleted versions of an object to be retained. This number of deleted versions, called the *retention count*, can be changed for any particular object. When this number is exceeded by the creation of another new version of an object, the oldest version of the object is destroyed.

More information about versions, including deleting and undeleting versions, specifying versions, and changing the retention count, can be found in Library Management (LM).

Committing Images

Objects within the directory system are permanent. For each object being edited, however, the Rational Editor internally creates temporary copies that are not part of the directory system. These temporary copies disappear if the session ends or if the system is shut down.

The temporary, presumably changed, copy remains temporary until the changes in the edited image are *committed*, thus making them permanent and a part of the directory system. Committing an image of an object can be done explicitly using the Common.Commit procedure, or it can be done implicitly by several operations such as the Common.Promote and Common.Release procedures.

Note that some image types do not require explicit committing because any changes made to them are immediately committed. Editing searchlists is one example of this.

Histories

The Rational Editor maintains histories of Command window image types. These histories allow changes to be undone, going back in time through each set of changes to the beginning of time for the image. Change histories can be stepped forward or backward with the Common.Redo and Common.Undo procedures, respectively.

Locks

Locks are used to prevent simultaneous updates to objects by different agents and to prevent updates when others are viewing objects. These locks are created by the Rational Editor and other Environment tools under various conditions. Locks can be created for all objects that exist in the directory system and for certain other entities in the Environment.

There are two kinds of locks: read only and write. A *read-only lock* is created when an object is viewed. A *write lock* is created when an object is made modifiable.

Updating Images

The images in windows are derived from underlying representations of the objects or information being edited. At times these underlying representations may change (for example, as the result of a job manipulating a file or adding a new unit to a library).

The Rational Editor attempts to keep the images up to date with respect to these changes. All images visible on the screen are automatically updated to reflect changes in the underlying representations whenever a job in the user's session finishes. An implication of this redraw algorithm is that, when a new window is manually brought onto the screen, its image may not be current.

The user can also cause images to be refreshed explicitly by executing the Common.Revert command. In addition, some type-specific editing commands cause images to be updated implicitly.

Library Switches

Some of the characteristics of the type-specific editing operations described in this section can be tailored using library switches. This tailoring includes:

- Specifying the case of identifiers in formatted Ada images.
- Specifying the format of output from jobs in text windows.

See LM, package Switches, for more information on library switches.

package Common

This package defines the commands that pertain to all type-specific editing operations. These commands typically are bound to keys (see the Keymap in Volume 1 of the *Rational Environment Reference Manual*). These commands perform the correct operation for the type of image in the current window.

procedure Abandon

```
procedure Abandon (Window : String := "<IMAGE>");
```

Description

Abandons editing of the current image and does not save changes.

This procedure abandons any changes made to the image in the current window since the last commit, releases any locks held by the Rational Editor on the entity corresponding to the image, destroys the window, and removes the image from the Window Directory.

When some objects are edited, those objects are locked so that no other user can edit them. This procedure releases the locks and destroys the window but does not commit the object first. It is similar to the Release procedure except that any changes since the last commit on the object are not saved.

Specifically, this procedure has the following effects:

- *Ada images:* Ends the editing of the Ada image. Any changes made to the image since the last commit or promote are lost. However, incremental changes made to installed or coded units, which are permanent as soon as they are promoted, are not lost. The window is removed from the screen and from the Window Directory. The Window parameter specifies the window to be removed from the screen. The default is the current image.
- *Command images:* Ends the editing of the command and removes the Command window from the screen. The Window parameter specifies which window should be removed from the screen. The default is the current image, which is the window above which or on which the command is executed.
- *Debugger:* Deletes the Debugger window if the Debugger has been killed. Otherwise, the command has no effect. This command has the same effect as the Release procedure.
- *Help and job windows:* Removes the window created by the !Commands.What-.Does, !Commands.What.Command, and !Command.What.Jobs procedures from the screen. This command has the same effect as the Release procedure.
- *Library images:* 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 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.
- *Links:* 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.

- *Menu images:* Ends the editing of the menu. The window is removed from the screen and from the Window Directory. This command has the same effect as the Release procedure. The Window parameter specifies which image to be abandoned. The default is the current image.
- *Searchlists:* Ends the editing of the searchlist and removes the window from the screen. Since all changes to searchlists are done immediately, this procedure does not abandon any of those changes.
- *Switches:* 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.
- *Text images:* Ends the editing of the image. Any changes made to the image since the last implicit or explicit commit are lost. The window is removed from the screen and from the Window Directory. The Window parameter specifies which window should be removed from the screen. The default, "<IMAGE>", removes the current image.

  If a job is currently performing input or output on an I/O window, the Abandon procedure will fail.
- *Window images:* Ends the editing of the Window Directory by removing the Window Directory from the screen. The Window parameter specifies the window to remove, which is, by default, the current image. This command has the same effect as the Release procedure.
- *Xref images:* Ends the editing of the xref. The window is removed from the screen and from the Window Directory. This command has the same effect as the Release procedure.

Parameters

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

Specifies the window that should be abandoned. The default is the current image.

References

procedure Commit

procedure Release

---

procedure Clear_Underlining

```
procedure Clear_Underlining;
```

Description

Removes all underlined error designations in the current image.

These designations typically result from errors in the image, but they can also result from running other commands that use underlining to indicate specific locations in the image—for example, the Ada.Show_Usage command.

Specifically, this procedure has the following effects:

- *Ada images:* Removes the underlining created by the Semanticize and other procedures. The Semanticize procedure checks the image for semantic consistency, underlining semantic errors.
- *Command images:* Removes all underlines in the current Command window.

procedure Commit

```
procedure Commit;
```

Description

Makes permanent any changes made to the image in the current window.

Specifically, this procedure has the following effects:

- *Ada images:* Makes permanent any changes to the Ada image. When source Ada images are edited, this procedure saves the changes to the image in the underlying permanent representation. These changes are built in temporary areas until the changes are committed. Then the temporary areas are made a permanent part of the storage hierarchy when a new version of the unit containing the changes is created.

  This procedure is used only for Ada images that are in the source state. Direct editing changes (that is, not incremental) to an Ada image in a state other than the source state are not allowed. Changes to the unit caused by promoting or demoting the unit are permanent and need not be committed.

  The commit operation is also implicitly performed by the Promote, Ada.Install_Unit, Ada.Code_Unit (if the unit is source, the operation is not incremental, and the operation completes successfully), and Release procedures.
- *Command images:* Executes the command in the Command window by formatting, semanticizing, and coding it. This procedure is identical to the Promote procedure.
- *Links:* Has no effect, because all changes to links are made immediately. All other operations on links implicitly commit any changes.
- *Searchlists:* Commits changes to the searchlist. Since all changes to searchlists are made immediately and permanently, this procedure has no effect. All other operations on searchlists implicitly commit any changes.
- *Switches:* 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.
- *Text images:* Makes permanent any changes to the image. Changes to the image are made in a temporary area. This procedure saves those changes, making them permanent, by creating a new version of the text file that contains the changes.

  The procedure also commits input in I/O windows. The input, along with a terminator or delimiter if necessary, is sent to the program that requested it. The Window parameter specifies which window's image should be committed. The default, "<IMAGE>", commits the current image.

- *Window images:* Makes permanent any changes to the image corresponding to the line designated in the Window Directory by executing the Common.Commit command on that image. If there is no selection, the procedure executes the Common.Commit command on all uncommitted images that are not I/O windows.

References

procedure Promote

---

procedure Complete

```
procedure Complete (Menu : Boolean := True);
```

Description

Completes the design ated item by inserting new text and prompts into the image using information about the syntax and semantics of the image type.

Specifically, this procedure has the following effects:

- *Ada images:* Completes the selected Ada identifier or the identifiers in the selected element using Ada's semantics for name resolution. If more than one name can complete the identifier and the Menu parameter is set to true, a list of choices is produced in the menu window. See the description of menus in this book for more information on the editing operations available on menus.

  The identifier is completed using the Ada context that exists for the compilation of the selected element, including ***with*** clauses, ***use*** clauses, renaming declarations, and so on.

- *Command images:* Completes the Ada fragment designated by the cursor using Ada semantics for name resolution. If more than one name could complete the identifier, and the Menu parameter is true, a list of choices is produced in the menu window. See the description of menus in this book for more information on the editing operations available on menus.

  This procedure completes only names of identifiers that are declared in the fragment in the Command window or that are visible through the searchlist. See package Search_List in Session and Job Management (SJM) for more information on searchlists.

  Declarations in Command windows must be selected before the Complete command will actually complete them. Statements need not be selected for the Complete command to complete them.

- *Library images:* Refreshes the library image in the current window to the current value of the underlying permanent representation and realigns the columns of the image.

Parameters

```
Menu :  Boolean := True;
```

Specifies whether a menu should be displayed for ambiguous references. The default, true, specifies that a menu will be displayed.

procedure Create_Command

```
procedure Create_Command;
```

Description

Creates a new Command window, if one does not already exist, before the current image, or it puts the cursor in the existing Command window that is closest to the current image.

Command windows can be created for Command windows recursively. A Command window acting on another Command window can be used to create or edit commands or to perform another activity from the same context without disturbing the contents of the existing Command windows.

Command windows have *use* clauses that are automatically added by the Create_Command procedure to provide direct visibility to the commands that are supported for the image type for which the Command window has been created. *Use* clauses for packages !Commands.Editor and Common are always present. Additional *use* clauses will be added based on the specific type of image for which the Command window has been created.

Command windows can be created for all image types.

Specifically, this procedure has the following effects:

- *Ada images:* Creates a Command window below the current Ada window if one does not exist; otherwise, it puts the cursor in the existing Command window below the current Ada window. This Command window initially has a *use* clause:

  ```
  use Editor, Ada, Common, Debug;
  ```

  This *use* clause provides direct visibility to the declarations in packages Ada, Common, Debug, and Editor without requiring qualification for names resolved in the command.

- *Command images:* Creates a Command window below the current Command window if one does not exist; otherwise, it puts the cursor in the existing Command window below the current Command window. This Command window initially has a *use* clause:

  ```
  use Editor, Command, Common;
  ```

  This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, Common, and Command without requiring qualification for names resolved in the command.

  For certain other window types, other packages are added to the *use* clause to provide visibility to operations used to edit those kinds of windows.

- *Debugger:* Creates a Command window below the Debugger window if one does not exist; otherwise, the command puts the cursor in the existing Command window below the Debugger window. This Command window initially has a *use* clause:

```
use Editor, Common, Debug;
```

  This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, Common, and Debug without requiring qualification for names resolved in the command.

- *Help and job windows:* Creates a Command window below the Help window or set of jobs if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the Help window or What.Jobs display. This Command window initially has a *use* clause:

```
use Editor, Common;
```

  This use clause provides direct visibility to the declarations in packages Editor and Common for names resolved in the command.

- *Library images:* 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 !Commands.Editor, Library, and Common without requiring qualification for names resolved in the command.

- *Links:* 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.

- *Menu images:* Creates a Command window below the menu if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the menu. This Command window initially has a *use* clause:

```
use Editor, Common;
```

  This *use* clause provides direct visibility to the declarations in packages !Commands.Editor and Common for names resolved in the command.

- *Searchlists:* Creates a Command window below the current window. The *use* clause in the Command window includes package Search_List, so operations in package !Commands.Search_List are visible in the Command window without qualification. The *use* clause for searchlist windows is:

```
use Editor, Search_List, Common;
```

- *Switches:* 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.

- *Text images:* Creates a Command window below the text window if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the text window. This Command window initially has a *use* clause:

  ```
  use Editor, Text, Common;
  ```

  This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, Text, and Common for names resolved in the command.

- *Window images:* Creates a Command window below the Window Directory if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the Window Directory. This Command window initially has a *use* clause:

  ```
  use Editor, Common;
  ```

  This *use* clause provides direct visibility to the declarations in packages !Commands.Editor and Common for names resolved in the command.

- *Xref images:* Creates a Command window below the xref if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the xref. This Command window initially has a *use* clause:

  ```
  use Editor, Ada, Common;
  ```

  This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, Ada, and Common for names resolved in the command.

---

procedure Definition

```
procedure Definition (Name     : String  := "<CURSOR>";
                      In_Place : Boolean := False;
                      Visible  : Boolean := True);
```

Description

Finds the defining occurrence of the named or designated item and displays that defining occurrence in a new window.

This procedure finds the location where the item is defined. The procedure attempts to find the most reasonable definition of the object, given the current editing context.

Specifically, this procedure has the following effects:

- *Ada images:* Finds the defining occurrence of the designated element and brings up its image in a window on the screen, typically with the definition of the element selected. If a name is provided to the Name parameter, it is used. If no name is provided, the cursor location is used to designate the element. A read-only lock is acquired on the unit.

  The In_Place parameter specifies whether the current frame should be used. The default is false. The Visible parameter specifies whether the specification or body should be displayed. The default, true, specifies that the specification should be preferred.

  The procedure finds the most reasonable definition of the element, given the current editing context. If the element is:

  — A subunit stub or an insertion point, the corresponding subunit is viewed (if the Visible parameter is false).

  — The visible part of a unit, the corresponding body of the unit is viewed (if the Visible parameter is false).

  — The body of a unit, the corresponding visible part of the unit is viewed (if the Visible parameter is true).

  — A usage of an identifier, the identifier's defining occurrence is viewed (if the Visible parameter is true or false).

  — A declaration of an object, the object's type declaration is viewed (if the Visible parameter is true or false).

  If the selected or designated item is in a subsystem spec view, then the default session activity is used to find its definition. See Project Management (PM) for more information on subsystems and activities.

  The following tables illustrate some additional examples of the use of the Definition command with the Visible parameter true or false. The first column specifies the location of the cursor when the Definition command is executed. The second

and third columns specify the effect of setting the Visible parameter to true or false, based on the position of the cursor.

Example 1:

```
package P1 is new P; -- instantiation of generic
     . . .
     P1.B;
```

Table 7-1. Instantiations

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
P1.B (cursor on P1)	instantiation	instantiation
P1.B (cursor on B)	generic specification	generic body

Example 2:

```
type T; -- incomplete type declaration
     . . .
type T is new integer; -- corresponding full type declaration
     X:T;
     . . .
```

Table 7-2. Incomplete Types

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
X:T (cursor on T)	full type declaration	full type declaration
full type declaration	incomplete type declaration	incomplete type declaration
incomplete type declaration	full type declaration	full type declaration

Example 3:

```
type T is private;
     . . .
type T is new integer;
     . . .
     X:T;
```

Table 7-3. Private Types

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
X:T (cursor on T)	private declaration	private part
private part	private declaration	private declaration
private declaration	private part	private part

Example 4:

```
task type T; -- task incomplete type
. . .
task type T is -- specification
. . .
task body T is
. . .
    X:T;
```

Table 7-4. Task Types

Cursor is on:	*Definition (Visible=>True) goes to:*	*Definition (Visible=>False) goes to:*
T1:T (cursor on T)	task specification	task body
entry call	task specification	task body
task specification	task body	task body
task incomplete type	task specification	task body

- *Command images:* Finds the defining occurrence of the named or designated element and brings up its image in a window on the screen, typically with the definition of the element selected. The Semanticize or Promote procedure must have been executed on the window containing the named or designated element. If a name is provided for the Name parameter, it is used. If no name is provided, a selection is used if one exists. Otherwise, the cursor location is used to designate the element.

  The procedure finds the most reasonable definition of the element, given the current editing context and the value of the Visible parameter. The Name parameter specifies which element's definition should be given. The In_Place parameter specifies whether the current window should be used, and the Visible parameter specifies whether the specification or the body should be preferred.
- *Debugger:* Finds the defining occurrence of the designated element and brings up its image in a window on the screen, typically with the definition of the element selected.
- *Help and job windows:* Brings up on the screen, for help windows, an image of the Ada specification unit containing the designated declaration in a Help window declaration. This procedure has no effect on displays created by What.Jobs.
- *Library images:* 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).
- *Links:* 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.

- *Menu images:* Brings up on the screen an image of the Ada compilation unit containing the designated declaration. The Name parameter specifies which image to display. The In_Place parameter specifies whether the current window should be used. The Visible parameter specifies whether the specification or body should be displayed.
- *Searchlists:* Finds the definition of the component in the searchlist to which the cursor points. This procedure creates a window containing that component. The In_Place parameter allows the user to specify whether the new window will replace the current window. The Visible parameter specifies whether the specification or the body should be preferred.
- *Switches:* 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.
- *Window images:* Moves the cursor to the image of the currently designated line, bringing that image onto the screen if necessary. The Name parameter specifies which image should be displayed. By default, it is the image corresponding to the current cursor location on the Window Directory. The In_Place parameter specifies whether the current window should be used to display the image. By default, the Window Directory is the next window replaced. The Visible parameter specifies whether the specification or the body should be displayed. The default, true, displays the specification.
- *Xref images:* Displays on the screen an image of the designated compilation unit with all usages of the declaration indicated with underlines. If the current level of detail is either views or subsystems, the procedure brings the image for the designated library onto the screen with no underlining.

  The cursor can be moved between underlined usages with the !Commands.Editor-.Cursor.Next and Editor.Cursor.Previous commands. The usage indications can be removed with the Clear_Underlining command.

  Some of the units in the xref display may implicitly depend on a declaration but may not have direct usages of the declaration. Also, the Environment is conservative about finding all possible units that depend on the declaration. Sometimes it accidentally includes a unit in the xref that has no reference. In these cases, executing the Definition command when designating such a unit deletes the unit from the xref and gives the message:

  ```
  <unit name> doesn't have references!  Zapping the line.
  ```

  Executing the Semanticize procedure searches each unit in the xref and deletes any entries that have no usages.

Parameters

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

Specifies the item for which to get the defining occurrence. The default is the item on which the cursor is currently located.

```
In_Place : Boolean := False;
```

Specifies whether the current frame should be used to bring up the image. The default specifies that the least recently used frame should be used.

```
Visible : Boolean := True;
```

Specifies how names that resolve to both a specification (visible part) and a body should be resolved. When true, the default, this parameter specifies that the specification is preferred. When false, it specifies that the body should be brought up, if possible.

References

procedure Edit

---

procedure Demote

```
procedure Demote;
```

Description

Demotes the designated item to a lower state.

Specifically, this procedure has the following effects:

- *Ada images:* Demotes an Ada unit or element to a lower unit state. If there is no selection or if the current selection is for an entire compilation unit, the procedure changes the state of the Ada unit in the current window, assuming there are no dependent units. If there are dependent units, a list of them is displayed in the menu window that is brought onto the screen. See the description of menus in this book 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.

  If there is a selection other than the entire unit and if incremental compilation is allowed on the element selected (see the rules on incremental compilation stated above), this procedure removes the element from the parent unit, replaces it with an insertion point, and leaves the element in the source state attached to the insertion point. The source for the element can be visited later by viewing the insertion point.

- *Command images:* Ends the editing of the command. The contents of the Command window are destroyed and the original contents are restored. All history is lost.

- *Library images:* 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 Menus, in this book, 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.

- *Menu images:* Attempts to demote the Ada unit containing the designated declaration to the next lower state. The specific effect of this procedure depends on the current state of the unit and whether other units depend on the unit on which the demote is being attempted. If there are no dependents and 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.

  If there are dependents, they are indicated by overwriting of the existing menu with a new menu containing these dependencies.
- *Text images:* Changes the current text window from read only to editable. If another user or job has a write lock on the file being viewed, the Demote command will fail. The procedure has no effect on I/O windows.
- *Window images:* Executes the Common.Demote procedure on the designated image.
- *Xref images:* Executes the Common.Demote procedure for Ada images on the selected unit.

---

procedure Edit

```
procedure Edit (Name     : String  := "<IMAGE>";
                In_Place : Boolean := False;
                Visible  : Boolean := False);
```

Description

Creates a writable image of the named or designated item, creating the window if necessary.

This procedure creates a new window that contains an image of the item to be edited. If a window with the image already exists, it is reused. The window is created with the default window size and is placed by the Rational Editor. The window remains in the Window Directory until the object is released or abandoned. The window disappears if it contains withdrawn or incrementally inserted declarations when the declarations or statements are promoted successfully.

Specifically, this procedure has the following effects:

- *Ada images:* Creates a window in which to edit the named or selected Ada unit and demotes the unit to source if necessary.

  If there is no selection or if the current selection is for an entire compilation unit or subunit declaration, the procedure creates a window in which to edit the unit, if necessary, and demotes the unit to source if no units depend on the unit. If there are dependent units, a list of them is displayed in the menu window that is brought onto the screen, and the operation fails. See the description of menus in this book for more information on the editing operations available on menus. If the operation succeeds, a write lock is acquired on the unit.

  If there is a selection other than the entire unit and if incremental compilation is allowed on the element selected (see the rules on incremental compilation stated above), this procedure removes the element from the parent unit, replaces it with an insertion point, and brings up a new window with the element in it.

  The Name parameter specifies the unit to edit. The default is `"<IMAGE>"`. The In_Place parameter specifies whether the current window should be used. The Visible parameter specifies whether the specification or body should be preferred.
- *Command images:* Replaces the contents of the Command window with a `[statement]` prompt.
- *Library images:* 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).
- *Links:* 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.

- *Menu images:* Creates a window in which to edit the Ada unit containing the selected declaration. The procedure demotes the unit to the source state, if necessary. If the demotion of the unit will cause obsolescence, the edit fails and a new menu of dependent units replaces the existing menu contents. The Name parameter specifies which unit to edit. The In_Place parameter specifies whether the current frame should be used. The Visible parameter specifies whether the specification or body should be displayed.
- *Searchlists:* Creates a Command window below the searchlist and places in it the command:

```
Replace (New_Component => "current value",
         Old_Component => "",
         Session => "",
         User => "");
```

The New_Component parameter allows the user to specify the new searchlist component. The Old_Component parameter specifies the searchlist component to be replaced. The user and session parameters specify the User and Session whose searchlist should be modified. The user must have read access to another user's home world to modify that user's searchlist.

- *Switches:* 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.

- *Text images:* Makes the current text window editable by acquiring a write lock on the file associated with the window. If other users or jobs have write locks on the file, the operation will fail. The procedure has no effect on I/O windows.

  The Name parameter specifies which text window should be made editable. The default special name, "<IMAGE>", specifies the current image or selection in a library image (if there is one). The In_Place parameter specifies whether the current frame should be used. The default, false, specifies that the current frame should not be used.

- *Window images:* Moves the cursor to the image of the currently designated image, bringing that window onto the screen if necessary. If the type of the designated image discriminates between viewing using the Definition command and editing using the Edit command, the procedure performs the operations associated with executing the Edit command on the designated image, unless these operations have already been performed on the image.

The Name parameter specifies which image should be displayed. The default is the image on which the cursor is located. The In_Place parameter specifies whether the current window should be used to display the image. The default is false. The Visible parameter specifies whether the specification or the body should be displayed. The default, true, displays the specification.

Parameters

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

Specifies the item to be edited. The default is the current image.

```
In_Place :  Boolean := False;
```

Specifies whether the current frame should be used to bring up the image. The default specifies that the least recently used frame should be used.

```
Visible :  Boolean := False;
```

Specifies how names that resolve to both a specification (visible part) and a body should be resolved. When true, the default, this parameter specifies that the specification is preferred. When false, it specifies that the body should be brought up, if possible.

---

procedure Elide

```
procedure Elide (Repeat : Positive := 1);
```

Description

Reduces the level of detail displayed in the image for the currently designated item.

This procedure accepts argument prefixes when executed from the keyboard to enable multiple levels of detail to be eliminated in a single operation.

Specifically, this procedure has the following effects:

- *Help and job windows:* Selects which set of jobs is displayed in the window. The procedure steps the display from all jobs, to all running jobs, to the user's jobs, to the user's running jobs, to all commands, to all running commands, to the user's commands, to the user's running commands. The default is all running jobs. This procedure has no effect on Help windows.
- *Library images:* Reduces the level of detail displayed for the designated object(s).
- *Links:* 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.
- *Menu images:* Decreases the level of detail displayed for the selected declaration to the next lower level. If no declaration is selected, the procedure decreases the level of detail for all of the declarations in the menu to the next lower level. The levels of detail available, ordered from lowest to highest, are:
  - — Simple names
  - — Full names
  - — Simple names with parameter profiles (the default)
  - — Full names with parameter profiles

  These levels are not circular; that is, expanding has no effect once the highest level of detail has been reached, and eliding has no effect once the lowest level of detail has been reached.
- *Switches:* 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.
- *Xref images:* Reduces the level of detail displayed in the current xref. This command is the opposite of the Expand command.

Although, by default, an xref displays the full name for each using unit, other levels of detail can be displayed using the Elide and Expand commands. These options are:

— Full_Names: Displays the full names of each unit with attributes (the default).

— Objects: Displays the unit name with attributes.

— Views: Displays the views using the declaration.

— Subsystems: Displays the subsystems using the declaration.

Executing the Elide command moves the level of detail down the above list; executing the Expand command moves the level of detail up the above list. The list is circular, so if you attempt to move down past the bottom, you go to the top; if you move up past the top, you go to the bottom. The current level of detail for an xref image is indicated in the banner for the xref.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of levels of detail to be eliminated. The default is 1.

References

procedure Expand

---

procedure Enclosing

```
procedure Enclosing (In_Place : Boolean := False;
                     Library  : Boolean := False);
```

Description

Finds the parent or enclosing item of the image in the current window and displays that item in a new window.

Specifically, this procedure has the following effects:

- *Ada images:* Finds the parent or enclosing Ada unit of the current window and displays that parent unit in a window. This procedure acquires a read lock on the unit. The In_Place parameter specifies whether the current window should be used. The Visible parameter specifies whether the specification or body should be preferred. The Library parameter specifies whether the resulting image should be a library.
- *Command images:* Finds the major window to which the Command window is attached and puts the cursor in it. The In_Place parameter specifies whether the current window should be used. The Library parameter specifies whether the enclosing object should be a library.
- *Debugger:* Displays the library containing the Command window from which the job being debugged was started.
- *Library images:* 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.
- *Links:* 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.
- *Switches:* 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.
- *Text images:* Brings up a window that contains an image of the library containing the file corresponding to the current text window. For I/O windows, the Enclosing procedure finds the home library for the user.

  The In_Place parameter specifies whether the current frame should be used. The default, false, specifies that the current frame should not be used. The Library parameter specifies whether the enclosing library should be displayed. The default is false.
- *Xref images:* Displays the library containing the unit for which the xref was created, with the unit selected.

Parameters

```
In_Place : Boolean := False;
```

Specifies whether the current frame should be used to bring up the image. The default specifies that the least recently used frame should be used.

```
Library : Boolean := False;
```

Specifies whether the resulting image should be a library. That is, for Ada subunits, this specifies that the enclosing library, rather than the parent body, should be displayed. The default is false.

Restrictions

The parent of the world ! is itself.

---

procedure Expand

```
procedure Expand (Repeat : Positive := 1);
```

Description

Increases the level of detail displayed in the image for the currently designated item.

This procedure accepts argument prefixes when executed from the keyboard to enable multiple levels of detail to be added in a single operation.

Specifically, this procedure has the following effects:

- *Help and job windows:* Selects which set of jobs is displayed in the window. The procedure steps the display from the user's running commands, to the user's commands, to all running commands, to all commands, to the user's running jobs, to the user's jobs, to all running jobs, to all jobs. The default is all running jobs. This procedure has no effect on Help windows.
- *Library images:* Increases the level of detail displayed for the designated object(s).
- *Links:* 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).
- *Menu images:* Increases the level of detail displayed for the selected declaration to the next higher level. If no declaration is selected, the procedure expands the level of detail for all of the declarations in the menu to the next higher level. The levels of detail available, ordered from lowest to highest, are:
  - — Simple names
  - — Full names
  - — Simple names with parameter profiles (the default)
  - — Full names with parameter profiles

  These levels are not circular; that is, expanding has no effect once the highest level of detail has been reached, and eliding has no effect once the lowest level of detail has been reached.
- *Switches:* 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.
- *Xref images:* Increases the level of detail displayed in the current xref. This command is the opposite of the Elide command.

Although, by default, an xref displays the full name for each using unit, other levels of detail can be displayed using the Expand and Elide commands. These options are:

— Full_Names: Displays the full names of each unit with attributes (the default).

— Objects: Displays the unit name with attributes.

— Views: Displays the views using the declaration.

— Subsystems: Displays the subsystems using the declaration.

Executing the Elide command moves the level of detail down the above list; executing the Expand command moves the level of detail up the above list. The list is circular, so if you attempt to move down past the bottom, you go to the top; if you move up past the top, you go to the bottom. The current level of detail for an xref image is indicated in the banner for the xref.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of levels of detail to be added. The default is 1.

References

procedure Elide

---

procedure Explain

```
procedure Explain;
```

Description

Provides explanatory information regarding the designated item in the current window.

Specifically, this procedure has the following effects:

- *Ada images:* Provides an explanation of the error designated by the cursor position in the Ada unit in the current window. Used after syntactic or semantic errors have been discovered, the procedure displays an explanation of those errors in the Message window.
- *Command images:* Provides an explanation of errors in the command in the current window. Used after syntactic or semantic errors have been discovered, the procedure displays an explanation of those errors in the Message window.
- *Help and job windows:* Adds an entry to the Help window for the designated item in a Help window menu. This procedure has no effect on displays created by What.Jobs.
- *Library images:* 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.
- *Links:* 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.
- *Switches:* Inserts, below the current switch, an explanation of that switch. If an explanation is already there, this procedure will remove it.
- *Xref images:* Displays the full name of the currently designated unit in the Message window.

procedure Format

```
procedure Format;
```

Description

Formats the current image appropriately for its image type.

For Ada units, this procedure checks the syntax of the image, performs syntactic completion, and pretty-prints again.

If changes are incomplete fragments, this procedure provides syntactic completion and prompting based on the syntax rules for the image type, and then it pretty-prints the image again with these changes and completions aligned and capitalized properly.

Specifically, this procedure has the following effects:

- *Ada images:* Formats the text in the current window. The procedure redraws some or all of the image after checking for syntactic errors and correcting or prompting for some of the syntactic constructs. If there are syntax errors in the image that cannot be corrected, they are marked as errors.

  This procedure adds ending punctuation, including semicolons, right parenthesis, closing quotation marks, *end loop* statements, *end if* statements, and *end* statements for packages and subprograms. Note that *end* statements are usually placed as close to the end of the source as is legal.

  Some of the behavior of the commands for editing Ada images can be tailored with session switches.

  The case of identifiers in formatted Ada images is determined by the Keyword_Case library switch. Allowable values for this switch are Upper, Lower, and Capitalize.

  See package Switches in Library Management (LM) for more information on library switches.

  Example 1:

  Before the Format procedure:

  ```
  procedure Push is begin
  ```

  After the Format procedure:

  ```
  procedure Push is
  begin
      [statement]
  end Push;
  ```

Example 2:

Before the Format procedure:

```
if case when
```

After the Format procedure:

```
if [expression] then
    case [expression] is
        when [expression] =>
            [statement]
    end case;
end if;
```

- *Command images:* Formats the text in the current Command window. The procedure redraws some or all of the image after checking for syntactic errors and correcting or prompting for some of the syntactic constructs. If there are syntactic errors in the image that cannot be corrected, they are marked as errors.

  Some of the behavior of the commands for editing commands can be tailored with session switches.

  The case of identifiers in formatted Ada images is determined by the Keyword_Case library switch. Allowable values for this switch are Upper, Lower, and Capitalize. Only one of these switches should have the value of true. The switch that is true determines how identifiers are displayed after the Format operation.

  See package Switches in Library Management (LM) for more information on session switches.

  Example 1:

  Before the Format procedure:

```
declare
    use Editor, Common, Ada;
begin
    if
end;
```

  After the Format procedure:

```
declare
    use editor, Common, Ada;
begin
    if [expression] then
        [statement]
    end if;
end;
```

- *Library images:* 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 Revert procedure.

procedure Insert_File

```
procedure Insert_File (Name : String := "<REGION>");
```

Description

Inserts the named text file into the current Ada image at the current cursor position.

No semantic analysis of the contents of the file or the resulting object is done.

Specifically, this procedure has the following effects:

- *Ada images:* Copies the contents of the text file specified in the Name parameter into the current Ada image at the current cursor position.
- *Command images:* Copies the contents of the named text file into the Command window at the current cursor position.
- *Text images:* Inserts the named text file into the current Ada image at the current cursor position.

Parameters

```
Name : String := "<REGION>";
```

Specifies the text to be inserted. The string can be a filename or a special name. The default is the current region.

procedure Promote

```
procedure Promote;
```

Description

Promotes the designated item to the next higher state.

Specifically, this procedure has the following effects:

- *Ada images:* Promotes the Ada image in the current window to the next higher state. The procedure changes the state of the Ada unit. The specific effect of this procedure depends on the current 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: Execution is attempted if the unit is selected. If parameters are required, the prompt for them appears in the Command window.

  If the current window is associated with an insertion point created by incremental compilation and the elements in the window are to be inserted in-line in the parent unit, this procedure causes the elements in the window to be inserted in the parent and the window is deleted.
- *Command images:* Executes the command by formatting, semanticizing, and coding it. This command is the same as the Commit procedure.
- *Library images:* 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.
- *Menu images:* Promotes the Ada unit containing the designated declaration to the next higher state. This procedure has the same effect as executing the Common.Promote command on the Ada unit containing the designated declaration. The specific effect of this procedure depends on the current 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: The procedure has no effect.

- *Switches:* 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.
- *Text images:* Commits changes to the image and releases the write lock on the underlying file. Changes to the image are made in a temporary area. This procedure saves those changes, making them permanent, by creating a new version of the text file that contains the changes.

  The procedure also commits input in I/O windows. The input, along with a terminator or delimiter if necessary, is sent to the program that requested it.
- *Window images:* Executes, on the image corresponding to the selected line, the Common.Promote procedure specific to that image type. If the image promoted is of Ada type and semantic errors are found, the image of the promoted unit is brought onto the screen with the errors underlined.
- *Xref images:* Executes the Common.Promote procedure for the selected Ada images on the xref.

References

procedure Commit

---

procedure Redo

```
procedure Redo (Repeat : Positive := 1);
```

Description

Redoes the Repeat changes previously made to an image.

The Rational Editor maintains histories of the temporary copies of some image types as images of these types are changed. These histories allow changes to be undone, going back in time through each set of changes to the beginning of time for the image. Change histories can be stepped forward or backward with the Redo and Undo procedures, respectively.

Change histories are retained by the Rational Editor, depending on the type of image. These histories are destroyed or restarted when the session ends and at other times, depending on the type of image being edited.

The opposite of this procedure is the Undo procedure.

Specifically, this procedure has the following effects:

- *Command images:* Recalls commands entered in a Command window after the Undo procedure is executed on the Command window. The Rational Editor remembers changes made to command images since the Command window was created. Each execution of the Complete, Edit, Demote, and Revert procedures marks another change, as well as execution of the command. The Repeat parameter specifies the number of commands to move forward in the history.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of changes to be redone. The default is the last set of changes.

References

procedure Undo

procedure Release

```
procedure Release (Window : String := "<IMAGE>";
```

Description

Ends editing on the current image and makes changes permanent.

This procedure releases any locks the Rational Editor may have in the entity being edited, destroys the window, and removes the image from the Window Directory.

This command does an implicit commit of the image.

Specifically, this procedure has the following effects:

- *Ada images:* Ends the editing of the Ada unit. The unit is unlocked, and any changes to the image are committed (made permanent). This window specified by the Window parameter is removed from the screen and from the Window Directory.
- *Command images:* Ends the editing of the command. The Command window is destroyed and removed from the screen. All history is lost. The Window parameter specifies the window to be released.
- *Debugger:* Deletes the Debugger window if the Debugger has been killed. Otherwise the command has no effect. This command has the same effect as the Abandon procedure.
- *Help and job windows:* Removes the window from the screen for the What.Jobs display. For Help windows, this procedure removes the Help window from the screen and from the Window Directory. This procedure has the same effect as the Abandon procedure.
- *Library images:* Ends the editing of the library image. The library image window is removed from the screen and from the Window Directory.
- *Links:* Ends the editing of the current set of links. The window is removed from the screen.
- *Menu images:* Ends the editing of the menu. The window is removed from the screen and from the Window Directory. This command has the same effect as the Abandon procedure. The Window parameter specifies which window to release. The default is the current image.
- *Searchlists:* Ends the editing of the searchlist and removes the window from the screen.
- *Switches:* Commits changes and ends the editing of the switches. The window is removed from the screen after any changes to the switches are saved.
- *Text images:* Ends the editing of the text and removes the image from the Window Directory. All changes to the text are made permanent before the window is

removed from the screen. A new version of the underlying file is created if changes are saved.

If a job is currently performing input or output on an I/O window, the Release procedure will fail.

The Window parameter specifies which window should be released. The default is the current image.

- *Window images:* Ends the editing of the Window Directory by removing the Window Directory from the screen. The Window parameter specifies the window to be removed. The default is the current image. This command has the same effect as the Abandon procedure.
- *Xref images:* Ends the editing of the xref image. The window is removed from the screen and from the Window Directory. This command has the same effect as the Abandon procedure.

Parameters

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

Specifies the window that should be released. The default is the current image.

References

procedure Abandon

procedure Commit

---

procedure Revert

```
procedure Revert;
```

Description

Restores the image in the current window to the current committed (permanent) value of the entity being edited and discards any changes that may have been made to the image.

Specifically, this procedure has the following effects:

- *Ada images:* Reverts the Ada image in the current window to the current value of the underlying permanent representation.
- *Command images:* Redraws the command in the current window.
- *Help and job windows:* Redraws the set of jobs to reflect the current state for the What.Jobs display. This procedure has no effect on Help windows.
- *Library images:* 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.
- *Links:* 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.
- *Searchlists:* Redraws the searchlist in the current window. If the searchlist has been changed by another user or program, this procedure redraws the list to ensure that the image is up to date.
- *Switches:* 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.
- *Text images:* Refreshes the image in the current window with the current value of the underlying file. Note that, if a job is writing into a file and the file is concurrently being viewed with the Rational Editor, the Revert command can be used to update the image to show any new output that has occurred since the last Revert procedure.
- *Window images:* Refreshes the image of the Window Directory so that the entries in it are current.

procedure Semanticize

```
procedure Semanticize;
```

Description

Checks the image in the current window to ensure that it is correct according to the syntax and semantic rules for the type of the image and indicates any errors.

Specifically, this procedure has the following effects:

- *Ada images:* Checks the Ada unit for semantic correctness. The procedure checks for compliance with the semantic rules of the Ada language. Errors discovered during semantic checking are underlined.
- *Command images:* Checks the command for semantic correctness. The procedure checks for compliance with the semantic rules of the Ada language. Errors discovered during semantic checking are underlined. This procedure must be executed either directly or indirectly (by using the Promote procedure) before the Definition procedure will execute successfully.
- *Xref images:* Searches each unit in the xref for actual usages and deletes any entries for units with no usages.

References

procedure Explain

procedure Format

procedure Promote

procedure Sort_Image

```
procedure Sort_Image (Format : Integer := 1);
```

Description

Sorts the display according to the given format.

Format numbering is specific to the object type. It is assumed that when the Format parameter is assigned the default value of 1, the display is sorted by increasing values. When the Format parameter has a value of –1, the image will be formatted in decreasing value.

Specifically, this procedure has the following effects:

- *Links:* 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.

Parameters

```
Format : Integer := 1;
```

Specifies the order in which the image will be sorted. The default value, 1, specifies that the display should be sorted in increasing value. A value of –1 specifies that the image should be sorted in decreasing value.

procedure Undo

```
procedure Undo (Repeat : Positive := 1);
```

Description

Undoes the previous Repeat sets of changes to the current image.

The Rational Editor maintains histories of the temporary copies of some image types as images of these types are changed. These histories allow changes to be undone, going back in time through each set of changes to the beginning of time for the image. Change histories can be stepped forward or backward with the Redo and Undo procedures, respectively.

Change histories are retained by the Rational Editor, depending on the type of image. These histories are destroyed or restarted when the session ends and at other times, depending on the type of image being edited.

The opposite of this procedure is the Redo procedure.

Specifically, this procedure has the following effects:

- *Command images:* Recalls changes previously made to a command. The Rational Editor remembers changes made to command images since the Command window was created, called a history. Each execution of the Edit, Demote, and Revert procedures marks another change, as well as each execution of the command. As commands are undone, the last undone command in the history becomes the place where new user-entered commands are saved in the history. These changes can be reinstated with the Redo procedure. The Repeat parameter specifies the number of commands to move backward in the history.
- *Library images:* Undeletes the selected object. This procedure is similar to the !Commands.Library.Undelete procedure.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of changes to be undone. The default, 1, is the last set of changes.

References

procedure Redo

procedure Write_File

```
procedure Write_File (Name : String := ">>FILE NAME<<");
```

Description

Writes the contents of the current selection into the named file.

If no there is no selection, this procedure writes the contents of the current image into the named file.

Specifically, this procedure has the following effects:

- *Command images:* Copies the contents of the selection in the Command window to the file specified by the Name parameter.
- *Debugger:* Writes the current contents of the Debugger window into the named file.
- *Text images:* Writes the contents of the current selection in the named file. If there is no selection, this procedure writes the contents of the current image into the named file. The previous contents of the file are lost.

Parameters

```
Name : String;
```

Specifies the file into which the current selection is to be written. The default parameter placeholder ">>FILE NAME<<" must be replaced or an error will result.

package Object

Package Object contains procedures for making selections based on the underlying type-specific structures of images and performing basic editing operations (including moving and copying) on these selections. For information on selecting regions of text and for procedures for selecting portions of images as text, see EI, package Editor.Region.

procedure Child

```
procedure Child (Repeat : Positive := 1);
```

Description

Selects the child of the designated item Repeat number of times, each time selecting the child of the child just selected.

The child is the item at the next lower level, in a syntactic sense, from the current item. The child that encloses the cursor is selected unless no such child exists.

Specifically, this procedure has the following effects:

- *Ada images:* Selects the Repeat child element of the currently selected element. A child element is one of the elements at the next lower level, in a syntactic sense, from the currently selected element. If an object at that level has not been selected before, the smallest element enclosing the cursor is chosen. If an element at that level has been selected before, the selection is turned off.
- *Command images:* Selects a child element of the currently selected element. A child element is one of the images at the next lower level, in a syntactic sense, from the current element. If an element at that level has not been selected before, the element on which the cursor is currently located is chosen. If an element at that level has been selected before, the selection is turned off.
- *Debugger:* Selects the Repeat child element of the currently selected element. A child element is one of the elements at the next lower level, in a syntactic sense, from the currently selected element. If an object at that level has not been selected before, the smallest element enclosing the cursor is chosen. If an element at that level has been selected before, it is selected again.
- *Help and job windows:* Selects the job on the line on which the cursor is located for the What.Jobs display, if one or more jobs are already selected when the procedure is entered. If there is a selection, the procedure leaves the current line selected. For Help windows, if the whole Help window is selected, the procedure selects the declaration on the line on which the cursor is located. If there is a selection, it leaves the current line selected.
- *Library images:* 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.
- *Links:* 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.
- *Menu images:* Selects the declaration on the line on which the cursor is located.

- *Searchlists:* Selects the component in the searchlist on which the cursor is located. If all components are already selected, the procedure selects the component on which the cursor is located. If a single component is already selected, the procedure has no effect. If no component is selected, the procedure selects the component on which the cursor is located.
- *Switches:* 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.
- *Text images:* Selects the next lower-level item in the hierarchical structure of a text image. The item selected will be the one the cursor is in if such an item exists. If no items are selected, the word closest to the cursor is selected.

  The Repeat parameter specifies the number of levels to move down in selecting the image. The default, 1, specifies the next lowest level.
- *Window images:* Selects the line on which the cursor is located, when the entire image is selected. When a single line is selected, the line is still selected. When nothing is selected, the procedure does nothing.
- *Xref images:* Selects the unit on the line on which the cursor is located.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of times the child selection is repeated.

References

procedure Next

procedure Parent

procedure Previous

---

procedure Copy

```
procedure Copy;
```

Description

Copies the selected item to the cursor position.

Specifically, this procedure has the following effects:

- *Ada images:* Copies the selected element to the cursor position. The new copy is in the source state. No semantic analysis is done on the selection in its new location, although a check is performed to ensure that a declaration is put in a declarative region and a statement is put in a statement region. Contained units of the copied element are not copied.
- *Library images:* 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.
- *Links:* 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.
- *Switches:* 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.
- *Text images:* Copies the selected text to the cursor position.

References

procedure Move

procedure Delete

```
procedure Delete;
```

Description

Deletes the designated item.

Specifically, this procedure has the following effects:

- *Ada images:* Deletes the selected element. If other elements are dependent on the element because of semantic references (from installed or coded units), the deletion fails and a menu of the dependent units is displayed in the menu window. See the description of the editing operations on menus in this book for more information. Contained units of the selected element are not deleted. The cursor must be in the selection for the operation to succeed.
- *Help and job windows:* Kills the selected job or the job on which the cursor is located for the What.Jobs display. If a job is selected, that job is deleted (terminated). If no job is selected, the job on which the cursor is located is deleted. Note that if the job is not for the current session and user, the command will fail. In the What.Jobs display, Object.Delete is equivalent to Job.Kill. This procedure has no effect on Help windows.
- *Library images:* 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 this book. Contained units of the selected element are not deleted. The cursor must be in the selection for the operation to succeed.
- *Links:* Deletes the selected link.
- *Searchlists:* Deletes from the searchlist the selected component or the component on which the cursor is located.
- *Switches:* Deletes the selected switch or the switch on which the cursor is located. A deleted switch assumes a system-defined default value.
- *Text images:* Deletes the selected text.
- *Window images:* Performs the Release command on the image described by the selected line. This causes any changes to the image to be made permanent and the selected line to be removed from the Window Directory. If no line is selected, the procedure fails, producing an error message.

procedure First_Child

```
procedure First_Child (Repeat : Positive := 1);
```

Description

Selects the first child of the designated item Repeat number of times.

The first child is the first one of the set of items at the next lower level, in a syntactic sense, from the current item.

Specifically, this procedure has the following effects:

- *Ada images:* Selects the first child of the currently selected element. The first child is the first one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.
- *Command images:* Selects the first child of the currently selected element. The first child is the first one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.
- *Debugger:* Selects the first child of the currently selected element. The first child is the first one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.
- *Help and job windows:* Selects the first job of the set for the What.Jobs display. For Help windows, this procedure selects the first line of the Help window.
- *Library images:* 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.
- *Links:* Selects the first link in the set of links.
- *Menu images:* Selects the first declaration in the menu.
- *Searchlists:* Selects the first component in the searchlist.
- *Switches:* Selects the first switch of the set.
- *Text images:* Selects the first child of the current selection. The first child is the first one at the next lower level in the hierarchy of the current selection.

  The Repeat parameter specifies the number of levels to move down in selecting the image. The default, 1, specifies the next lowest level.
- *Window images:* Selects the first line of the Window Directory.
- *Xref images:* Selects the first unit in the xref.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of times the selection is repeated.

References

procedure Last_Child

---

procedure Insert

```
procedure Insert;
```

Description

Enables the user to insert a new item.

Specifically, this procedure has the following effects:

- *Ada images:* Creates an insertion point in installed and coded units where statements, declarations, other elements on which incremental compilation operations are supported, or an entire compilation unit can be inserted into the current element.
- *Library images:* Creates an insertion point in a library where an Ada compilation unit can be inserted.
- *Links:* 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.
- *Searchlists:* Creates a Command window and places in it the command:

  ```
  Add (Component => "[STRING-expression]",
       Position => 1,
       Session => "",
       User => "");
  ```

  where the first parameter is a string that can specify one or more components (separated with commas) and the second parameter is the position within the searchlist. Providing a value for the first parameter and promoting the command adds the specified component to the searchlist. The Session and User parameters allow the user to specify another session or username's searchlist to which to add an entry.
- *Switches:* 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.

- ***Window images:*** Creates a new Command window below the Window Directory and prompts for the Definition command as follows:

```
Definition ("");
```

procedure Last_Child

```
procedure Last_Child (Repeat : Positive := 1);
```

Description

Selects the last child of the designated item Repeat number of times.

The last child is the last one of the set of items at the next lower level, in a syntactic sense, from the current item.

Specifically, this procedure has the following effects:

- *Ada images:* Selects the last child of the currently selected element. The last child is the last one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.
- *Command images:* Selects the last child of the currently selected element. The last child is the last one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.
- *Debugger:* Selects the last child of the currently selected element. The last child is the last one of the set of elements at the next lower level, in a syntactic sense, from the currently selected element.
- *Help and job windows:* Selects the last job of the set for the What.Jobs display. For Help windows, this procedure selects the last line of the Help window.
- *Library images:* 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.
- *Links:* Selects the last link of the set.
- *Menu images:* Selects the last declaration in the menu.
- *Searchlists:* Selects the last component in the searchlist.
- *Switches:* Selects the last switch of the set.
- *Text images:* Selects the last child of the current selection. The last child is the last one at the next lower level in the hierarchy of the current selection.

  The Repeat parameter specifies the number of levels to move down in selecting the image. The default, 1, specifies the next lowest level.
- *Window images:* Selects the last line in the Window Directory.
- *Xref images:* Selects the last unit in the xref.

Parameters

```
Repeat :  Positive := 1;
```

Specifies the number of times the selection is repeated.

References

procedure First_Child

---

procedure Move

```
procedure Move;
```

Description

Moves the selected item to the cursor position.

This procedure moves the current selection to the cursor position by copying the item and then deleting the original item.

Specifically, this procedure has the following effects:

- *Ada images:* Moves the selected element to the cursor position. This movement is done by copying the element and then deleting the original element. The new copy is placed in the source state. If other elements are dependent on the element because of semantic references (from installed or coded units), the deletion fails but the copy succeeds. Contained units of the selected unit are not moved.
- *Library images:* 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.
- *Links:* 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.
- *Searchlists:* Moves the selected searchlist component to the current cursor position in the current searchlist.
- *Switches:* 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.
- *Text images:* Moves the selected text to the cursor position.

procedure Next

```
procedure Next (Repeat : Positive := 1);
```

Description

Selects the next item past the designated item Repeat number of times.

The next item is the item at the same level, in a syntactic sense, as the designated item that appears immediately after the designated item.

Specifically, this procedure has the following effects:

- *Ada images:* Selects the Repeat next element past the currently selected element. A next element is the element at the same level, in a syntactic sense, as the current element that appears immediately after the current element. If no such selection can be made, the next element at the enclosing level is selected.
- *Command images:* Selects the Repeat next element past the currently selected element. A next element is the element at the same level, in a syntactic sense, as the current element that appears immediately after the current element. If no such selection can be made, the next element at the enclosing level is selected.
- *Debugger:* Selects the Repeat next element past the currently selected element. A next element is the element at the same level, in a syntactic sense, as the current element that appears immediately after the current element. If no such selection can be made, the next element at the enclosing level is selected.
- *Help and job windows:* Selects the job that is listed after the currently selected job, provided that the cursor is on currently selected job, for the What.Jobs display. If the cursor is not on the currently selected job or if no job is already selected, the procedure selects the job on which the cursor is located. If all jobs are selected, this procedure produces an error message.

  For Help windows, this procedure selects the next item past the current item declaration if the cursor is in the current selection; otherwise, it selects the item corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the item corresponding to the line on which the cursor is located. If the entire Help window is selected, this procedure produces an error message.
- *Library images:* Selects the next object at the same or greater level past the currently selected object.
- *Links:* 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.
- *Menu images:* Selects the next declaration past the currently selected declaration if the cursor is in the current selection; otherwise, the procedure selects the declaration corresponding to the line on which the cursor is located. If nothing is

currently selected, the procedure selects the declaration corresponding to the line on which the cursor is located. The Repeat parameter specifies that the Repeat declaration after the currently selected declaration is to be selected.

- *Searchlists:* Selects the next component in the searchlist. If no component is already selected, the procedure selects the component on which the cursor is located. If all components are selected, this procedure produces an error.
- *Switches:* 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.
- *Text images:* Selects the next item after the current selection at the same level in the text-image hierarchy. If there is no current selection, the word after the current cursor position is selected.

  The Repeat parameter specifies the number of the selection to be selected after the current cursor position.
- *Window images:* Selects the next line past the currently selected line if the cursor is in the current selection; otherwise, the procedure selects the line on which the cursor is located. If nothing is currently selected, the procedure selects the line on which the cursor is located. The Repeat parameter specifies to select the Repeat line after the currently selected line.
- *Xref images:* Selects the next unit past the currently selected unit if the cursor is in the current selection; otherwise, the procedure selects the unit corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the unit corresponding to the line on which the cursor is located.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of times the selection is repeated.

References

procedure Child

procedure Parent

procedure Previous

---

procedure Parent

```
procedure Parent (Repeat : Positive := 1);
```

Description

Selects the parent item of the designated item Repeat number of times.

The parent is the item at the next higher level, in a syntactic sense, from the designated item that contains the designated item.

Specifically, this procedure has the following effects:

- *Ada images:* Selects the parent element of the currently selected element. The parent element is the element that contains the current element at the next higher level, in a syntactic sense, from the current element.
- *Command images:* Selects the parent element of the currently selected element. The parent element is the element that contains the current element at the next higher level, in a syntactic sense, from the current element.
- *Debugger:* Selects the parent element of the currently selected element. The parent element is the element that contains the current element at the next higher level, in a syntactic sense, from the current element.
- *Help and job windows:* Selects the job the cursor is on for the What.Jobs display. If a job is already selected and the cursor is on the currently selected job, the procedure selects all jobs in the set. If all jobs are already selected, the procedure has no effect.

  For Help windows, this procedure selects the item corresponding to the line on which the cursor is located if there are no selections; otherwise, it selects the entire Help window. If the entire Help window is already selected, the procedure has no effect.
- *Library images:* 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.
- *Links:* 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.
- *Menu images:* Selects the declaration corresponding to the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire menu.
- *Searchlists:* Selects the component in the searchlist on which the cursor is located. If no component is already selected, the procedure selects the component on which the cursor is located. If a component is selected, the procedure selects all components in the set. Otherwise, the procedure has no effect.

- *Switches:* 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.
- *Text images:* Selects the next higher-level item in the hierarchical structure of a text image. The item selected is the one in which the cursor or current selection is located. If no items are selected, the word closest to the cursor is selected.

  The Repeat parameter specifies the number of levels to move up in selecting the image. The default, 1, specifies the next highest level.
- *Window images:* Selects the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire Window Directory.
- *Xref images:* Selects the unit corresponding to the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire list of units in the xref.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of times the selection is repeated.

References

procedure Child

procedure Next

procedure Previous

---

procedure Previous

```
procedure Previous (Repeat : Positive := 1);
```

Description

Selects the previous item before the designated item Repeat number of times.

The previous item is the item at the same level, in a syntactic sense, as the designated item that appears immediately before the designated item.

Specifically, this procedure has the following effects:

- *Ada images:* Selects the Repeat previous element before the currently selected element. A previous object is the object at the same level, in a syntactic sense, as the current element that appears immediately before the current element. If no such selection can be made, the previous element at the enclosing level is selected.
- *Command images:* Selects the Repeat previous element before the currently selected element. A previous element is the element at the same level, in a syntactic sense, as the current element that appears immediately before the current element. If no such selection can be made, the previous element at the enclosing level is selected.
- *Debugger:* Selects the Repeat previous element before the currently selected element. A previous object is the object at the same level, in a syntactic sense, as the current element that appears immediately before the current element. If no such selection can be made, the previous element at the enclosing level is selected.
- *Help and job windows:* Selects the job that is listed before the currently selected job, provided that the cursor is on the currently selected job, for the What.Jobs display. If the cursor is not on the currently selected job or if no job is already selected, the procedure selects the job on which the cursor is located. If all jobs are selected, this procedure produces an error.

  For Help windows, the procedure selects the previous item before the currently selected item, if the cursor is in the current selection; otherwise, it selects the item corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the item corresponding to the line on which the cursor is located.
- *Library images:* Selects the previous object at the same or greater level before the currently selected object.
- *Links:* 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.
- *Menu images:* Selects the previous declaration before the currently selected declaration if the cursor is in the current selection; otherwise, the procedure selects the declaration corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the declaration corresponding to

the line on which the cursor is located. The Repeat parameter specifies that the Repeat declaration before the currently selected declaration is to be selected.

- *Searchlists:* Selects the previous component in the searchlist. If no component is already selected, the procedure selects the component on which the cursor is located. If all components are selected, this procedure produces an error.
- *Switches:* 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.
- *Text images:* Selects the previous item before the current selection at the same level in the text-image hierarchy. If there is no current selection, the word before the current cursor position is selected.

  The Repeat parameter specifies the number of the selection to be selected before the current cursor position.
- *Window images:* Selects the previous line before the currently selected line if the cursor is in the current selection; otherwise, the procedure selects the line on which the cursor is located. If nothing is currently selected, the procedure selects the line on which the cursor is located. The Repeat parameter specifies to select the Repeat line after the currently selected line.
- *Xref images:* Selects the previous unit before the currently selected unit if the cursor is in the current selection; otherwise, the procedure selects the unit corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the unit corresponding to the line on which the cursor is located.

Parameters

```
Repeat : Positive := 1;
```

Specifies the number of times the selection is repeated.

References

procedure Child

procedure Next

procedure Parent

---

end Object;

---

end Common;

Help

This section describes the on-line help facility for the Rational Environment. It also describes type-specific editing operations for Help windows.

The Help commands are part of package !Commands.What (see SJM, package What).

Organization of the On-Line Help Facility

The help facility supports three types of help accessible from keys: help on keys with [Help on Key], help on the help facility itself with [Help on Help], and help on commands with [Help]. The Help window can be displayed by pressing [Help Window].

All help messages are displayed in the Help window. The beginning of each help message is delineated by a series of dashes. Each time help is requested, the resulting help message is added to the end of the Help window. Thus, help messages are separated by dashed lines.

The following display illustrates the contents of a help message:

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

!COMMANDS.WHAT.TIME is bound to:  F20

 procedure Time;

 Returns the current date and time of day in the Message window.
```

Help messages contain the following parts:

- The first line of a help message, a dashed line, indicates the beginning of a help message.
- The name of the procedure and the names of any keys bound to the procedure are indicated below the dashed line. Note that the same procedure, with a different parameter profile, may be bound to different keys and thus will have different results. To determine if this is the case for a key in which you are interested, [Help on Key] displays the parameter profile in the Message window for the key you press.
- The next part of the message shows the Ada specification for the operation being explained. In this case, it is the line `procedure Time;`.

- The rest of the help message explains the declaration.

Each time help is requested, the new help message is added to the end of the Help window. Thus, you can review previous help messages, as described in the following section.

Reviewing Previous Help Messages

All help messages displayed in the Help window are saved for the current login session; in other words, all the help messages you request from the time you log in until you log out are saved in the Help window. When you log out, the contents of the Help window are no longer saved.

If you want to review a help message from your current login session, you can return to the Help window to look at the help message again. If the Help window is currently on the screen, you can move the cursor back into the Help window using the [Window]-[↑] or [Window]-[↓] combinations or by pressing [Help Window].

If the Help window has been removed from the screen, you can redisplay it by pressing [Help Window]. The Help window will be displayed and the cursor will be moved to that window. You can also redisplay the Help window by using the Window Directory.

Moving the Cursor in the Help Window

While the cursor is in the Help window, you can scroll through its contents using Cursor and Image commands (EI, packages Editor.Cursor and Editor.Images). Alternatively, you can use the search facilities to look for a particular entry (EI, package Editor.Search).

Menus in the Help Window

If a name you specify to the [Help] key (the !Commands.What.Does procedure is bound to this key) resolves to more than one entry in the help system, a menu will be displayed in the Help window listing all of the declarations for the related help messages. You can use designation, described below, to specify which declaration's help message you want to see. Then you can press [Explain] to display the help message for that menu item.

Designation

To specify a declaration displayed in a menu for which you want to see help information, you can use the following two forms of designation:

- Selection: You can select the declaration in a menu by using the selection commands in package !Commands.Common.Object or package !Commands.Editor-.Region. Note that, for the selection to be valid, the cursor must remain in the selection.
- Cursor position: To designate a declaration in a menu, put the cursor anywhere on the line corresponding to the declaration.

You can then press [Explain] to see the help message for the designated menu declaration.

Special Names

Help commands use special names. For further information on special names, see Key Concepts.

Getting Help on the Help Facility

The [Help on Help] key allows you to get help on a command or on the help facility itself. To get help on the help facility itself, press [Help on Help]. The Help window will display a help message for the help facility itself.

Getting Help on Keys

The [Help on Key] key describes the function of the next key you press. For example, to find out what [Promote] does:

1. Press [Help on Key]. The system prompts you in the Message window with:

   ```
   Press key to be described:
   ```

2. Press the key for which you want to see a Help message. In this example, you press [Promote].

Once you press the key, its name and parameter profile appear in the Message window. Next, the system displays the Help window. A help message for the key that you pressed is displayed in the Help window.

New help messages are appended to the end of the contents of the Help window. The last entry in the Help window will be the latest help requested—in this example, help on [Promote].

Getting Help Using Selection

The [Help] key allows you to use selection to get help on a call to a declaration if it appears in a window on your screen. For example, if you want help on the !Commands.Common.Promote procedure, and a call to it or its declaration appears in a window on your screen, follow these steps:

1. Select the call to the declaration or the declaration itself (use commands in package !Commands.Common.Object or !Commands.Editor.Region to make the selection).
2. Press [Help]. The system will display help for the selected item.

Note that the window containing a call will produce more satisfactory results if the window has been semanticized. If the window has not been semanticized, you can use the procedure outlined below in "Getting Help on Commands."

Getting Help on Commands

The [Help] key allows you to get help on a declaration, whether or not you know its fully qualified name or even its complete simple name. For example, if you want to find out about the Write procedure and you do not know its fully qualified name, follow these steps:

1. Press [Prompt For].
2. Press [Help]. In a Command window, the system prompts you with:

```
What.Does (Name => "");
```

3. Move the cursor to the first quotation mark, and type `write`. (The quotation marks remain.) When you have finished, the command is:

```
What.Does (Name => "write");
```

4. Press [Promote].

If only one entry in the help system has the word "write" in it, the system displays its help message in the Help window. On the other hand, if more than one entry is related to "write" (as in this case), the system displays a menu in the Help window containing the declarations for all the related entries. Thus, the menu for "write" might be:

```
--------------------------------------------------------------------------
!Commands.Access_List.Write
!Commands.Activity.Write
!Commands.Common.Write_File
!Commands.Switches.Write
!Commands.Tape.Write
!Commands.Tape.Write_Mt
!Io.Direct_Io.Write
!Io.Polymorphic_Sequential_Io.Operations.Write
!Io.Sequential_Io.Write
!Io.Window_Io.Overwrite
!Tools.Access_List_Tools.Write
```

You can use designation (described above) to designate the menu item whose help message you want to see, and then you can press [Explain] to display the help message. If you display a help message for a menu item in which you are not interested, you can always scroll back up to the menu to select a different menu item.

If you are interested in seeing the Ada specification for one of the procedures in the menu, you can designate its declaration in the menu or in its help message, and then press [Definition]. The system will display another window containing the Ada specification.

Getting Help on a Topic

If you want help on a topic and do not know the complete name of the corresponding declaration, you can use the procedure described in "Getting Help on Commands,"

above. For example, if you want to know how to change your password, you can request help with the command:

```
What.Does (Name => "password");
```

Getting Help Using a Command Window

If you know the fully qualified pathname for a command, you can use the following procedure to display help information for it:

1. Open a Command window by pressing [Create Command].
2. In the Command window, on the [statement] prompt, type:

   ```
   What.Does
   ```

3. Press [Complete]. The system displays the parameter profile for the command:

   ```
   What.Does (Name => "");
   ```

4. Move the cursor to the first quotation mark. Then type the name of the procedure. For example, if you want help on the !Commands.Common.Definition procedure, type:

   ```
   Common.Definition
   ```

   When you have finished, the command is:

   ```
   What.Does (Name => "Common.Definition");
   ```

   Note that you can simply type the name of the operation—for example, `definition`. If this resolves to more than one entry in the help system, a menu is displayed. You can use designation to select a declaration from the menu, and then press [Explain] to see the help message for that declaration.

5. Press [Promote] to execute the command.

The system displays the command's help message in the Help window.

Determining Key Bindings

You can determine the key to which a command is bound by using the procedure described above in "Getting Help on Commands." When the help system displays the help message for the command, it also displays the name of any key to which the procedure is bound.

Commands from Package Common

The following commands from package !Commands.Common are supported for the Help window. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Removes the Help window from the screen and from the Window Directory. This command has the same effect as the Release procedure.

procedure Common.Create_Command

Creates a Command window below the Help window if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the Help window. This Command window initially has a *use* clause:

```
use Editor, Common;
```

This *use* clause provides direct visibility to the declarations in packages !Commands.Editor and !Commands.Common for names resolved in the command.

procedure Common.Definition

Brings up on the screen an image of the Ada unit containing the designated declaration.

procedure Common.Explain

Adds an entry to the Help window for the designated declaration in a Help window menu.

procedure Common.Release

Removes the Help window from the screen and from the Window Directory. This command has the same effect as the Abandon procedure.

procedure Common.Object.Child

Selects the line the cursor is on if the whole Help window is selected. If there is a selection, the procedure leaves the current line selected.

procedure Common.Object.First_Child

Selects the first line in the Help window.

procedure Common.Object.Last_Child

Selects the last line in the Help window.

procedure Common.Object.Next

Selects the next line after the currently selected line if the cursor is in the current selection; otherwise, the procedure selects the line on which the cursor is located. If nothing is currently selected, the procedure selects the line on which the cursor is located.

procedure Common.Object.Parent

Selects the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire Help window.

procedure Common.Object.Previous

Selects the previous line before the currently selected line if the cursor is in the current selection; otherwise, it selects the line on which the cursor is located. If nothing is currently selected, the procedure selects the line on which the cursor is located.

Menus

This section describes type-specific editing operations for lists of Ada declarations generated by the !Commands.Common.Complete command and !Commands.Editor commands that attempt to demote Ada declarations. These lists, of menu image type (referred to in this section as *menus*), are created to describe:

- The possible declarations to which an incomplete Ada name fragment can resolve.
- The compilation units that become obsolete as a result of demoting or withdrawing/deleting (using the incremental compilation operations) an Ada declaration.

There is only one menu for a session. When a new menu is created, the current one is destroyed. These menus can be viewed at various levels of detail to help in determining the specific declaration referred to by the item in the menu. Various other operations are supported for declarations in menus, including viewing, editing, promoting, and demoting the Ada units in which the declarations appear. All of these operations come from package Common. This section describes the common commands as they pertain to editing menus. The common editing operations are discussed more fully in package Common in this book.

For more information on how menus are generated, see the reference entry for the Common.Complete command for images of types Ada and Command, as well as other commands that explicitly or implicitly demote Ada units.

Image Structure

Here are examples of two menus, one generated by attempting to demote the specification of package Complex in a library containing some other units that depend on it, and another generated while attempting to complete a statement in a Command window. They will be used in the discussions that follow.

The menu generated from demoting COMPLEX'Spec is:

```
Units that are obsolesced by COMPLEX'Spec

  Complex'body
  Complex_List'spec
  Complex_Utilities'spec
  Display_Complex_Sums'body
```

The menu generated from attempting to complete Text_Io.Put in a Command window is:

```
List of possible completions

  Put =>
       Put'spec
       Put'spec
       Put'spec
       Put'spec
       Put_Line'spec
       Put_Line'spec
```

The menu is composed of a title and a list of lines. Each line corresponds to an Ada declaration, which could be an entire compilation unit or an individual declaration visible in a compilation unit. By default, only the simple names of these declarations are displayed. However, this level of detail can be changed (see below).

There is only one menu for each session. If subsequent commands generate new menus—for example, if a menu is created when an attempt is made to edit a unit—old menu values are lost. Attempting to demote a declaration in the menu may cause another menu, listing the units that depend on this declaration, to be created and to overwrite the previous menu. This menu is referred to as the ***menu window.***

The menu has no name associated with it.

Key Concepts

Designation

Structurally, a menu is a list of declarations, one to a line. Each row of the table corresponds to a unit. To designate one of the declarations in the menu for an operation, you have three options:

- Put the cursor anywhere on the line corresponding to the declaration that you want.
- Select the line corresponding to the declaration that you want using one of the selection commands from package !Commands.Common.Object.
- Select the entire line using the !Commands.Editor.Region.Start and Editor.Region.Finish commands.

Although it is possible to select an entire menu, menus ignore such multiple-line selections. If the entire menu is selected, the location of the cursor is used to designate the declaration.

Note that, if there is a single-line image selection in the menu, this selection is used instead of the location of the cursor. You can turn off any existing selections anywhere in the image with the Editor.Region.Off command. If a multiple-line selection exists, the cursor position determines which declaration is displayed.

Special Names

Some of the commands in package Common use special names to indicate a designation, 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 highlighted text in the image 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.

Expansion and Elision

By default, the declarations in a menu are displayed with their simple names and parameter profiles. However, additional detail is available if desired. The !Commands.Common.Expand and Elide commands can be used to increase or decrease the level of detail displayed in a menu. These commands can be applied to a selected declaration, sequence of declarations, or, if none is selected, all of the declarations in the menu.

The levels of detail available, ordered from lowest to highest, are:

- Simple names
- Full names
- Simple names with parameter profiles (the default)
- Full names with parameter profiles

These levels are not circular; that is, expanding has no effect once the highest level of detail has been reached, and eliding has no effect once the lowest level of detail has been reached.

Here are examples of the differing levels of detail for the menu described previously, which resulted from attempting to complete an incomplete statement:

Simple names:

```
List of possible completions

Put =>
    Put'spec
    Put'spec
    Put'spec
    Put'spec
    Put_Line'spec
    Put_Line'spec
```

Full names:

```
List of possible completions

Put =>
    !IO.TEXT_IO.PUT'N(1)
    !IO.TEXT_IO.PUT'N(2)
    !IO.TEXT_IO.PUT'N(3)
    !IO.TEXT_IO.PUT'N(4)
    !IO.TEXT_IO.PUT_LINE'N(1)
    !IO.TEXT_IO.PUT_LINE'N(2)
```

Simple names with parameter profiles:

```
List of possible completions

Put =>
    procedure Put (File : File_Type; Item : Character);
    procedure Put (Item : Character);
    procedure Put (File : File_Type; Item : String);
    procedure Put (Item : String);
    procedure Put_Line (File : File_Type; Item : String);
    procedure Put_Line (Item : String);
```

Full names with parameter profiles:

```
List of possible completions

Put =>
    procedure !IO.TEXT_IO.PUT (File : File_Type; Item : Character);
    procedure !IO.TEXT_IO.PUT (Item : Character);'spec
    procedure !IO.TEXT_IO.PUT (File : File_Type; Item : String);
    procedure !IO.TEXT_IO.PUT (Item : String);'spec
    procedure !IO.TEXT_IO.PUT_LINE (File : File_Type; Item : String);
    procedure !IO.TEXT_IO.PUT_LINE (Item : String);
```

Commands from Package Common

The following commands from package !Commands.Common are supported for editing menus. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Ends the editing of the menu. The window is removed from the screen and from the Window Directory. This command has the same effect as the Release procedure. The Window parameter specifies which image to be abandoned. The default is the current image.

procedure Common.Create_Command

Creates a Command window below the menu if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the menu. This Command window initially has a *use* clause:

```
use Editor, Common;
```

This *use* clause provides direct visibility to the declarations in packages !Commands.Editor and !Commands.Common for names resolved in the command.

procedure Common.Definition

Brings up on the screen an image of the Ada compilation unit containing the designated declaration. The Name parameter specifies which image to display. The In_Place parameter specifies whether the current window should be used. The Visible parameter specifies whether the specification or body should be displayed.

procedure Common.Demote

Attempts to demote the Ada unit containing the designated declaration to the next lower state. The specific effect of this procedure depends on the current state of the unit and whether other units depend on the unit on which the demote is being attempted. If there are no dependents and 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.

If there are dependents, they are indicated by overwriting of the existing menu with a new menu containing these dependencies.

procedure Common.Edit

Creates a window in which to edit the Ada unit containing the selected declaration. The procedure demotes the unit to the source state, if necessary. If the demotion of the unit will cause obsolescence, the edit fails and a new menu of dependent units replaces the existing menu contents. The Name parameter specifies which unit to edit. The In_Place parameter specifies whether the current frame should be used. The Visible parameter specifies whether the specification or body should be displayed.

procedure Common.Elide

Decreases the level of detail displayed for the selected declaration to the next lower level. If no declaration is selected, the procedure decreases the level of detail for all of the declarations in the menu to the next lower level. The levels of detail available, ordered from lowest to highest, are:

- Simple names
- Full names
- Simple names with parameter profiles (the default)
- Full names with parameter profiles

These levels are not circular; that is, expanding has no effect once the highest level of detail has been reached, and eliding has no effect once the lowest level of detail has been reached.

procedure Common.Expand

Increases the level of detail displayed for the selected declaration to the next higher level. If no declaration is selected, the procedure expands the level of detail for all of the declarations in the menu to the next higher level. The levels of detail available, ordered from lowest to highest, are:

- Simple names
- Full names
- Simple names with parameter profiles (the default)
- Full names with parameter profiles

These levels are not circular; that is, expanding has no effect once the highest level of detail has been reached, and eliding has no effect once the lowest level of detail has been reached.

procedure Common.Promote

Promotes the Ada unit containing the designated declaration to the next higher state. This procedure has the same effect as executing the Common.Promote command on the Ada unit containing the designated declaration. The specific effect of this procedure depends on the current 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: The procedure has no effect.

procedure Common.Release

Ends the editing of the menu. The window is removed from the screen and from the Window Directory. This command has the same effect as the Abandon procedure. The Window parameter specifies which window to release. The default is the current image.

procedure Common.Object.Child

Selects the declaration on the line on which the cursor is located.

procedure Common.Object.First_Child

Selects the first declaration in the menu.

procedure Common.Object.Last_Child

Selects the last declaration in the menu.

procedure Common.Object.Next

Selects the next declaration past the currently selected declaration if the cursor is in the current selection; otherwise, the procedure selects the declaration corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the declaration corresponding to the line on which the cursor is located. The Repeat parameter specifies that the Repeat declaration after the currently selected declaration is to be selected.

procedure Common.Object.Parent

Selects the declaration corresponding to the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire menu.

procedure Common.Object.Previous

Selects the previous declaration before the currently selected declaration if the cursor is in the current selection; otherwise, the procedure selects the declaration corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the declaration corresponding to the line on which the cursor is located. The Repeat parameter specifies that the Repeat declaration before the currently selected declaration is to be selected.

Text Images

This section describes type-specific editing operations for text images. Text is composed of sequences of ASCII characters organized as lines of words that form sentences and paragraphs.

Text images are created in two different ways:

- When files in the directory system are viewed or edited (such images are referred to as *text*).
- When a job performs interactive input and output to a window (such images are referred to as *I/O windows*).

The power of the Rational Editor available for editing job input and output is basically the same as for editing files, but there is no permanent underlying object as there is with a file. When editing text images associated with jobs, the editor provides prompts for input and permits full editing of input and output text. However, the image in the text window is temporary and is destroyed when the session is terminated.

Note that the Rational Debugger is a program that performs interactive input and output to the Debugger window, so it is also an image of text type. See Debugging (DEB) for more information on the Rational Debugger.

Many operations in package !Commands.Common apply to text. These commands, as well as those in package !Commands.Text, are described in this section.

Image Structure

The structure of text images is a strict five-level hierarchy, consisting of (from the lowest level to the highest level):

- ASCII characters
- Words
- Sentences
- Paragraphs

- The whole text image

Note that the characters and words of a text image can be grouped into one or more lines, and they can be operated on with commands from package !Commands.Editor.Line.

Words are contiguous sequences of characters delimited by the set of word delimiters. By default, these delimiters include spaces, most punctuation characters, and the Ada delimiters. The delimiters are established on a per-session basis and can be changed either by executing the !Commands.Editor.Word.Breaks command or by editing the Word_Breaks session switch. See Editing Images (EI) for more information on word breaks and Session and Job Management (SJM) for more information on session switches.

Sentences consist of one or more words, delimited either by a blank line or by a sentence delimiter followed by two or more spaces. The sentence delimiters are the period (.), the question mark (?), and the exclamation (!).

Paragraphs consist of one or more sentences separated by blank lines.

There are always five levels in this hierarchy, even if only a single character. For example, if there is only a single character with an empty line above and below it, that character is a word, a sentence, a paragraph (this property can be useful when using selection, which is described below), and the entire image.

Key Concepts

Designation

Designation can be accomplished in one of two ways:

- By indicating the start and end of a selection using the !Commands.Editor.Region.Start and Editor.Region.Finish commands.
- By using the commands in package !Commands.Common.Object.

Using the selection operations from package Common.Object allows selection of lower- or higher-level elements in the hierarchical structure of text. The commands in package Common.Object can be used to select the word the cursor is in, the sentence the selected word is in, the paragraph the selected sentence is in, or the entire text image a selected paragraph is in. The commands can also be used to select a particular paragraph from a fully selected text image, a particular sentence from a selected paragraph, or a particular word from a selected sentence.

Note that, if there are no selections, selecting the parent of a character three times (with the Common.Object.Parent command) always selects the paragraph the cursor is in, even if the paragraph consists of only a single character, word, or sentence. This property can be useful in constructing higher-level commands and/or keyboard macros that perform operations on sentences and paragraphs without requiring the user to select the sentence or paragraph beforehand. (For example, a paragraph

fill operation that fills the paragraph the cursor is in can be implemented by performing the Common.Object.Parent operation three times and then executing the Editor.Region.Fill command on the selected paragraph.)

If there are no selections, selecting the parent of a character four times selects the entire image.

Versions

Versions of text files exist just as versions of other objects exist in the Environment. A new version of a text file is created when a file is committed or promoted.

Committing Images

Changes that are made to the text images corresponding to files do not become permanent until the changes are committed (saved). Committing can be done explicitly with the !Commands.Common.Commit and Common.Promote procedures or implicitly with the Common.Release procedure.

Locks

The Rational Editor creates a *write lock* on a text file when it is opened for editing. This lock allows only one user to edit the file, although users can view the file's contents even while someone is editing it (or while a job—for example, a compilation log—is writing to it). In such cases, the viewers of the locked file are warned that the file is currently open when they attempt to view it. The file remains locked until it is released or promoted.

If a file is being viewed—that is, the window containing the image was created with the Common.Definition command and/or the equals sign (=) is in the left end of the banner—another user or job can obtain a write lock on the file and the original viewers will not be notified.

Job Input and Output

When a job is inputting or outputting to a window, its job name/context and number appear in the banner for the window (and also in the Window Directory entry for the image). When the job terminates, the job number is removed, but the name/context entries stay until they are replaced with new values the next time a job uses the window.

When a job is requesting input from the window (for example, with the !Io.Text_Io-.Get command), an `[input]` prompt appears. Text can be entered at this prompt and can continue for multiple lines. The full power of the Rational Editor is available to cut and paste, change, and so on or to continue to add to this text. When the input has been properly composed, it can be sent to the program by executing the !Commands.Common.Commit, the Common.Promote, or the Common.Format command.

Executing the Common.Format command causes the text, exactly as entered, to be passed as input to the requesting job. Executing the Common.Enter or the

Common.Promote command provides a line terminator or delimiter if requested by the input operation and not supplied in the text entered at the [input] prompt.

When jobs output to windows, the text images they write are protected. This means that these areas of the text image cannot be modified by editing them. Note that, for I/O windows, the asterisk (*) and pound (#) symbols can appear in the banner. The # symbol means that a job is running that has requested input, but no input has been entered. When input is entered, the symbol changes from # to *. When this input is committed, the symbol changes back to #. When the job completes, the symbol changes to a blank. Because images with * and # are considered uncommitted, you cannot log off without terminating jobs requesting input unless you ignore changed images (the Debugger window is a special case that the Rational Editor treats as committed even though it has a # symbol).

The context for I/O windows is that of the last job to send output to or get input from that window.

Session Switches

Some of the behavior of job output to text windows can be tailored with session switches. (See LM, Session Switches, for more information on session switches.) Those switches that specifically pertain to text output displays typically begin with "Text_".

Commands from Package Common

The following commands from package !Commands.Common are supported for editing text. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Ends the editing of the image. Any changes made to the image since the last implicit or explicit commit are lost. The window is removed from the screen and from the Window Directory. The Window parameter specifies which window should be removed from the screen. The default, "<IMAGE>", removes the current image.

If a job is currently performing input or output on an I/O window, the Abandon procedure will fail.

procedure Common.Commit

Makes permanent any changes to the image. Changes to the image are made in a temporary area. This procedure saves those changes, making them permanent, by creating a new version of the text file that contains the changes.

The procedure also commits input in I/O windows. The input, along with a terminator or delimiter if necessary, is sent to the program that requested it. The Window parameter specifies which window's image should be committed. The default, "<IMAGE>", commits the current image.

procedure Common.Create_Command

Creates a Command window below the text window if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the text window. This Command window initially has a *use* clause:

```
use Editor, Text, Common;
```

This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, !Commands.Text, and !Commands.Common for names resolved in the command.

procedure Common.Demote

Changes the current text window from read only to editable. If another user or job has a write lock on the file being viewed, the Demote command will fail. The procedure has no effect on I/O windows.

procedure Common.Edit

Makes the current text window editable by acquiring a write lock on the file associated with the window. If other users or jobs have write locks on the file, the operation will fail. The procedure has no effect on I/O windows.

The Name parameter specifies which text window should be made editable. The default special name, "<IMAGE>", specifies the current image or selection in a library image (if there is one). The In_Place parameter specifies whether the current frame should be used. The default, false, specifies that the current frame should not be used.

procedure Common.Enclosing

Brings up a window that contains an image of the library containing the file corresponding to the current text window. For I/O windows, the Enclosing procedure finds the home library for the user.

The In_Place parameter specifies whether the current frame should be used. The default, false, specifies that the current frame should not be used. The Library parameter specifies whether the enclosing library should be displayed. The default is false.

procedure Common.Insert_File

Inserts the named text file into the current Ada image at the current cursor position.

procedure Common.Promote

Commits changes to the image and releases the write lock on the underlying file. Changes to the image are made in a temporary area. This procedure saves those changes, making them permanent, by creating a new version of the text file that contains the changes.

The procedure also commits input in I/O windows. The input, along with a terminator or delimiter if necessary, is sent to the program that requested it.

procedure Common.Release

Ends the editing of the text and removes the image from the Window Directory. All changes to the text are made permanent before the window is removed from the screen. A new version of the underlying file is created if changes are saved.

If a job is currently performing input or output on an I/O window, the Release procedure will fail.

The Window parameter specifies which window should be released. The default is the current image.

procedure Common.Revert

Refreshes the image in the current window with the current value of the underlying file. Note that, if a job is writing into a file and the file is concurrently being viewed with the Rational Editor, the Revert command can be used to update the image to show any new output that has occurred since the last Revert procedure.

procedure Common.Write_File

Writes the contents of the current selection in the named file. If there is no selection, this procedure writes the contents of the current image into the named file. The previous contents of the file are lost.

procedure Common.Object.Child

Selects the next lower-level item in the hierarchical structure of a text image. The item selected will be the one the cursor is in if such an item exists. If no items are selected, the word closest to the cursor is selected.

The Repeat parameter specifies the number of levels to move down in selecting the image. The default, 1, specifies the next lowest level.

procedure Common.Object.Copy

Copies the selected text to the cursor position.

procedure Common.Object.Delete

Deletes the selected text.

procedure Common.Object.First_Child

Selects the first child of the current selection. The first child is the first one at the next lower level in the hierarchy of the current selection.

The Repeat parameter specifies the number of levels to move down in selecting the image. The default, 1, specifies the next lowest level.

procedure Common.Object.Last_Child

Selects the last child of the current selection. The last child is the last one at the next lower level in the hierarchy of the current selection.

The Repeat parameter specifies the number of levels to move down in selecting the image. The default, 1, specifies the next lowest level.

procedure Common.Object.Move

Moves the selected text to the cursor position.

procedure Common.Object.Next

Selects the next item after the current selection at the same level in the text-image hierarchy. If there is no current selection, the word after the current cursor position is selected.

The Repeat parameter specifies the number of the selection to be selected after the current cursor position.

procedure Common.Object.Parent

Selects the next higher-level item in the hierarchical structure of a text image. The item selected is the one in which the cursor or current selection is located. If no items are selected, the word closest to the cursor is selected.

The Repeat parameter specifies the number of levels to move up in selecting the image. The default, 1, specifies the next highest level.

procedure Common.Object.Previous

Selects the previous item before the current selection at the same level in the text-image hierarchy. If there is no current selection, the word before the current cursor position is selected.

The Repeat parameter specifies the number of the selection to be selected before the current cursor position.

RATIONAL

package Text

This package contains the set of procedures and types provided for text object-specific editing. The commands in package !Commands.Common can also be used for text object-specific editing.

procedure Block

```
procedure Block  (All_Windows : Boolean := False);
```

Description

Temporarily stops all job output to the current window or to all windows, based on the value of the All_Windows parameter.

The output can be resumed with the Continue procedure. If the window is already blocked, this procedure has no effect.

If All_Windows is true, all I/O windows, not just the current one, are blocked.

The !Commands.Job.Disable or Job.Kill procedure can also be used to stop jobs (see SJM, package Job).

Parameters

```
All_Windows :  Boolean := False;
```

Specifies whether the current I/O window is blocked (the default) or all I/O windows are blocked.

References

procedure Continue

SJM, procedure Job.Disable

SJM, procedure Job.Kill

procedure Continue

```
procedure Continue (Page_Mode    : Boolean := False;
                    All_Windows  : Boolean := False);
```

Description

Resumes job output to the current window or to all windows that have been blocked using the Block procedure, depending on the value of the All_Windows parameter.

If the window is not blocked, this procedure has no effect. If the All_Windows parameter is true, all I/O windows, not just the current one, are continued.

If the Page_Mode parameter is true, the I/O window or windows automatically block after an additional page of output has been displayed. At that point, this procedure must be executed again to resume output.

Parameters

`Page_Mode : Boolean := False;`

Specifies whether the window automatically blocks again after the next page of output (true) or continues displaying output until explicitly blocked or completed (the default).

`All_Windows : Boolean := False;`

Specifies whether the current I/O window is continued (the default) or all I/O windows are continued.

References

procedure Block

procedure Create

```
procedure Create (Image_Name : String     := ">>IMAGE NAME<<";
                 Kind       : Image_Kind := Text.File);
```

Description

Creates a new empty text file in a new window for editing or for use by a job performing I/O.

This procedure can be used in two ways:

- It can be used to create a text file. The Image_Name parameter specifies the name of the file to be created. To create a text file, the Kind parameter must specify Text.File. A window is created containing the image of the text file when the procedure is executed.
- It can be called from a job to create an I/O window for that job. When used in a job, the Image_Name parameter specifies the name that will appear in the window of the job and the I/O window created by the job. The Kind parameter must specify Text.Input_Output so that an I/O window is created. No text file is created.

If the named file exists, its contents are destroyed. The new file created by this procedure is always empty.

If the kind of image created is a file, then the image name must be a legal filename.

Parameters

`Image_Name : String := ">>IMAGE NAME<<";`

Specifies the name of the image or file. If the kind of image is a file, then this string must be a legal filename. The default parameter placeholder `">>IMAGE NAME<<"` must be replaced or an error will result.

`Kind : Image_Kind := Text.File;`

Specifies the kind of image to be created. The default is a text file. To create an I/O window, specify Text.Input_Output.

References

type Image_Kind

procedure End_Of_Input

```
procedure End_Of_Input;
```

Description

Signals the interacting program that no more input will be provided, when executed in an I/O window.

The !Io.Io_Exceptions.End_Error exception is raised if the program requests more input.

type Image_Kind

```
type Image_Kind is (File, Input_Output);
```

Description

Defines the two kinds of text images.

This type is used by the Create procedure to specify what kind of text image to create.

Enumerations

`File`
Specifies that the procedure using this type will interact with the image of a text file.

`Input_Output`
Specifies that the procedure using this type will interact with an I/O image.

References

procedure Create

---

procedure Redirect

```
procedure Redirect (To : String := ">>File Name<<");
```

Description

Redirects the output associated with the current output window to the specified file.

This procedure is useful when you want to log off and save the contents of the current output window.

Parameters

```
To :  String := ">>File Name<<";
```

Specifies the name of the file to which output should be directed. If a fully qualified filename is not supplied, the name of the file is assumed to be in the same context as the job that initiated the output window.

end Text;

Window Directory

This section describes type-specific editing operations for the Window Directory, which is a windows image type. The Window Directory is a list of active images that can be manipulated with the Rational Editor.

The !Commands.Editor.Window.Directory procedure creates a window, called the ***Window Directory***, that contains a list of all currently active images and indicates their name, size, type, time of last modification, and whether they have been modified since last committed. (See EI, package Editor.Window, for more information on this procedure.)

A common operation in the Window Directory is to bring an image listed in the Window Directory onto the screen. This is done by moving the cursor to the line in the Window Directory for the desired image and executing the !Commands.Common.Definition command. Other operations, such as committing, can also be performed.

This section describes the commands from package !Commands.Common as they pertain to editing windows. The common editing operations are discussed more fully in the documentation for package Common in this book.

Image Structure

This typical Window Directory image will be used in the following discussion:

```
MOD  LINES       TYPE            BUFFER NAME
===  =====   ============    ================================
 =    5222   (text)          !USERS.BLB.MAKE_LOG'V(3)
 =       6   (menu)
 =      11   (searchlist)    <BLB,S_1>
 =       2   (switch)        !USERS.BLB.LIBRARY_SWITCHES'V(1)
 *       9   (ada)           !USERS.BLB.COMPLEX'V(4)
 =     122   (links)         !USERS.BLB
         7   (ada)           !USERS.BLB % UNIT_1(JOB 2Ø4)
 =      92   (library)       !USERS.BLB
         1                   Help Window
 =       8                   Message Window
 =      13   (windows)       Window Directory
```

Note that this image is a table of four columns with a row for each active image in a session. As new images are created, they are added to the top of the Window Directory.

The first column, MOD, is a flag indicating whether the image has been modified and/or formatted since the last time it was committed. This column also indicates whether the image is read only (read) or is modifiable (write). Note that this is the same symbol that appears in the lower-left corner of the banner of the image to which the entry corresponds. In general, the symbol may have slightly different meanings based on the type of the image. The following table provides the typical interpretation of these symbols:

Table 12-1. Interpretation of Symbols in the Window Directory

Symbol	*Lock*	*Committed?*	*Formatted?*
=	Read	(Not applicable)	True
(Blank)	Write	True	Can be either
*	Write	False	False
#	Write	False	True
!	(The editor or a job has temporarily locked the image)		

Note that, for I/O windows, the * and # symbols mean that a job that is requesting input is running. Because such images are considered uncommitted, you cannot log off without terminating these jobs or ignoring changed images.

The second column, LINES, indicates the number of text lines in the image.

The third column, TYPE, indicates the type of the image.

The fourth column, BUFFER NAME, indicates the name of the image. If the name is too long, it will be elided on the left. To determine the full name of an image, go to the image and press [What Object]. The full name of the image will be displayed in the Message window.

Key Concepts

Designation

The Window Directory structurally is a list of images, one to a line. Each row of the table corresponds to an image. To designate one of the images in the directory for an operation such a promote or a commit, you have three options:

- Put the cursor anywhere on the line corresponding to the image that you want.
- Select the line corresponding to the image that you want using one of the selection commands from package !Commands.Common.Object.
- Select the entire line (including any trailing blanks and, optionally, the line terminator) using the !Commands.Editor.Region.Start and Editor.Region.Finish commands.

Although it is possible to select the entire Window Directory or several lines of it, the Window Directory ignores such multiple-image selections. If the entire image is selected, the location of the cursor is used to designate the image.

Note that, if there is a single-image selection in the Window Directory, this selection is used instead of the location of the cursor. You can turn off any existing selections anywhere in the image with the Editor.Region.Off command.

Some of the commands in package !Commands.Common use special names to indicate a designation, 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 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.

Traversing to Images

The Window Directory allows you to view or edit any of the images it lists. To do so, first designate the image you want. Then execute the !Commands.Common.Definition or Common.Edit command to view or edit the unit. The execution of either of these commands causes the image for the unit to be brought onto the screen.

Committing, Promoting, and Demoting

The Window Directory allows you to commit, promote, or demote images listed in it. To do so, first select the image you want to operate on, and then execute the !Commands.Common.Commit, Common.Promote, or Common.Demote command. The command is performed on the selected image without the image being brought onto the screen.

Releasing Images

The Window Directory allows you to commit any changes that have been made to an image and to remove that image from the Window Directory without first bringing the image onto the screen. To do so, select the line that lists the image you want to delete with the commands from !Commands.Common.Object or !Commands.Editor.Region as described above, and then execute the !Commands.Common.Object.Delete command. This operation is equivalent to performing the Common.Release command on the selected image.

Refreshing the Window Directory

The image of the Window Directory that is displayed is usually current. Sometimes, however, the entries in the directory may not have been updated to reflect their current values. If you need the most current information, execute the !Commands.Common.Revert command to refresh the Window Directory image.

If an operation is attempted on an image appearing in the Window Directory image but not currently in the Window Directory (because the image has not been refreshed), the operation will have no effect and the image will be refreshed.

Commands from Package Common

The following commands from package !Commands.Common are supported for editing the Window Directory. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Ends the editing of the Window Directory by removing the Window Directory from the screen. The Window parameter specifies the window to remove, which is, by default, the current image. This command has the same effect as the Release procedure.

procedure Common.Commit

Makes permanent any changes to the image corresponding to the line designated in the Window Directory by executing the Common.Commit command on that image. If there is no selection, the procedure executes the Common.Commit command on all uncommitted images that are not I/O windows.

procedure Common.Create_Command

Creates a Command window below the Window Directory if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the Window Directory. This Command window initially has a *use* clause:

```
use Editor, Common;
```

This *use* clause provides direct visibility to the declarations in packages !Commands.Editor and Common for names resolved in the command.

procedure Common.Definition

Moves the cursor to the image of the currently designated line, bringing that image onto the screen if necessary. The Name parameter specifies which image should be displayed. By default, it is the image corresponding to the current cursor location on the Window Directory. The In_Place parameter specifies whether the current window should be used to display the image. By default, the Window Directory is the next window replaced. The Visible parameter specifies whether the specification or the body should be displayed. The default, true, displays the specification.

procedure Common.Demote

Executes the Common.Demote procedure on the designated image.

procedure Common.Edit

Moves the cursor to the image of the currently designated image, bringing that window onto the screen if necessary. If the type of the designated image discriminates between viewing using the Definition command and editing using the Edit command, the procedure performs the operations associated with executing the Edit command on the designated image, unless these operations have already been performed on the image.

The Name parameter specifies which image should be displayed. The default is the image on which the cursor is located. The In_Place parameter specifies whether the current window should be used to display the image. The default is false. The Visible parameter specifies whether the specification or the body should be displayed. The default, true, displays the specification.

procedure Common.Promote

Executes, on the image corresponding to the selected line, the Common.Promote procedure specific to that image type. If the image promoted is of Ada type and semantic errors are found, the image of the promoted unit is brought onto the screen with the errors underlined.

procedure Common.Release

Ends the editing of the Window Directory by removing the Window Directory from the screen. The Window parameter specifies the window to be removed. The default is the current image. This command has the same effect as the Abandon procedure.

procedure Common.Revert

Refreshes the image of the Window Directory so that the entries in it are current.

procedure Common.Object.Child

Selects the line on which the cursor is located, when the entire image is selected. When a single line is selected, the line is still selected. When nothing is selected, the procedure does nothing.

procedure Common.Object.Delete

Performs the Release command on the image described by the selected line. This causes any changes to the image to be made permanent and the selected line to be removed from the Window Directory. If no line is selected, the procedure fails, producing an error message.

procedure Common.Object.First_Child

Selects the first line of the Window Directory.

procedure Common.Object.Insert

Creates a new Command window below the Window Directory and prompts for the Definition command as follows:

```
Definition ("");
```

procedure Common.Object.Last_Child

Selects the last line in the Window Directory.

procedure Common.Object.Next

Selects the next line past the currently selected line if the cursor is in the current selection; otherwise, the procedure selects the line on which the cursor is located. If nothing is currently selected, the procedure selects the line on which the cursor is located. The Repeat parameter specifies to select the Repeat line after the currently selected line.

procedure Common.Object.Parent

Selects the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire Window Directory.

procedure Common.Object.Previous

Selects the previous line before the currently selected line if the cursor is in the current selection; otherwise, the procedure selects the line on which the cursor is located. If nothing is currently selected, the procedure selects the line on which the cursor is located. The Repeat parameter specifies to select the Repeat line after the currently selected line.

Xref Images

This section describes type-specific editing operations for xref images, which are lists of Ada compilation units using a particular declaration. These lists are created by the !Commands.Ada.Show_Usage command (see the description of this command in package Ada in this book).

Xrefs are created by the Ada.Show_Usage command. When the !Commands.Common.Definition command is executed on a compilation unit name listed in an xref, the image for the compilation unit is brought onto the screen and all usages of the declaration are indicated with underlines. The cursor can be moved between these usages with the !Commands.Editor.Cursor.Next and Editor.Cursor.Previous commands. The usage indications can be removed with the !Commands.Common-.Clear_Underlining command.

Xrefs can also be used to examine the subsystems and views that reference a declaration. See Project Management (PM) more information on subsystems and views.

The commands for editing xrefs are in package !Commands.Common. This section describes these package Common commands as they pertain to editing xrefs. The common editing operations are discussed more fully in package Common in this book.

Image Structure

Here is an xref image generated by showing the usage for the specification of package Complex in a small library containing some utilities and a main program built on package Complex. This xref image will be used in the following discussions:

```
!USERS.DEMO.DEVELOPMENT_EXAMPLE.COMPLEX'BODY
!USERS.DEMO.DEVELOPMENT_EXAMPLE.COMPLEX_LIST
!USERS.DEMO.DEVELOPMENT_EXAMPLE.COMPLEX_UTILITIES'BODY
!USERS.DEMO.DEVELOPMENT_EXAMPLE.COMPLEX_UTILITIES
!USERS.DEMO.DEVELOPMENT_EXAMPLE.DISPLAY_COMPLEX_SUMS'BODY
```

The xref is composed of a list of lines. Each line corresponds to a compilation unit that has been compiled and contains at least one reference to a declaration on the

specification for package Complex. By default, the names of these compilation units are fully qualified. This level of detail can be changed (see below).

The name of the xref is the name of the declaration for which the usage is being shown. In this example, the name is !Users.Demo.Development_Example.Complex. If this name is elided on the left in the banner, the !Commands.What.Object command can be used to get its full name.

Key Concepts

Designation

Structurally, an xref is a list of compilation units, one to a line. Each line corresponds to a compilation unit (unless the image has been expanded and/or elided; see below). To designate one of the units in the xref for operations such as definition, put the cursor anywhere on the line corresponding to the unit you want. Optionally, you can designate the unit using the selection commands from package !Commands.Common.Object or package !Commands.Editor.Region.

Expansion and Elision

Although, by default, an xref displays the full name for each using unit, other levels of detail are available through the !Commands.Common.Expand and Common.Elide commands. These options are:

- Full_Names: Displays the full names of each unit with attributes (the default).
- Objects: Displays the unit name with attributes.
- Views: Displays the views using the declaration.
- Subsystems: Displays the subsystems using the declaration.

Eliding moves the level of detail down the above list. Expanding moves the level of detail up the above list. The list is circular, so if you attempt to move down past the bottom, you go to the top; if you move up past the top, you go to the bottom. The current level of detail for an xref image is indicated in the banner for the xref.

Views and subsystems are two levels of detail that pertain primarily to use with Rational Subsystems™ (refer to Project Management (PM) for more information). If views and subsystems are used on nonsubsystem libraries, they have the following meanings. Views displays the libraries that contain units using the declaration. Subsystems displays the libraries containing the libraries that contain the units using the declaration. With either the views or the subsystems level of detail, the Common.Definition command brings the image for the designated library onto the screen.

The following images show the other levels of detail for the example xref described earlier:

Objects:

```
COMPLEX'BODY
COMPLEX_LIST
COMPLEX_UTILITIES'BODY
COMPLEX_UTILITIES
DISPLAY_COMPLEX_SUMS'BODY
```

Views:

```
!USERS.DEMO.DEVELOPMENT_EXAMPLE
```

Subsystems:

```
!USERS.DEMO
```

Unit States and False Usages

The !Commands.Ada.Show_Usage command can be used to find all usages of a declaration by units that have been promoted to unit states of installed or coded. It does not find usages in units that are in the source state.

Some of the units in the xref display may implicitly depend on a declaration but may not have direct usages of the declaration. Also, the Environment is conservative about finding all possible units that depend on the declaration. Sometimes it accidentally includes a unit in the xref that has no reference, usually because of overloading. In these cases, executing the !Commands.Common.Definition command when designating such units deletes the unit from the xref and gives the message:

```
<unit name> doesn't have references!  Zapping the line.
```

Executing the Common.Semanticize procedure searches each unit in the xref and deletes any entries that have no usages. (This can take a long time if there are many units in the xref or if the units in the xref are very large.)

Note that if the declaration is a package specification, its body and subunits will not be included in the xref unless they contain actual usages of the package name in the declarations and statements that compose them.

Commands from Package Common

The following commands from package !Commands.Common are supported for editing xrefs. If a command is not included in this list, it is not supported.

procedure Common.Abandon

Ends the editing of the xref. The window is removed from the screen and from the Window Directory. This command has the same effect as the Release procedure.

procedure Common.Create_Command

Creates a Command window below the xref if one does not exist; otherwise, the procedure puts the cursor in the existing Command window below the xref. This Command window initially has a *use* clause:

```
use Editor, Ada, Common;
```

This *use* clause provides direct visibility to the declarations in packages !Commands.Editor, !Commands.Ada, and !Commands.Common for names resolved in the command.

procedure Common.Definition

Displays on the screen an image of the designated compilation unit with all usages of the declaration indicated with underlines. If the current level of detail is either views or subsystems, the procedure brings the image for the designated library onto the screen with no underlining.

The cursor can be moved between underlined usages with the !Commands.Editor-.Cursor.Next and Editor.Cursor.Previous commands. The usage indications can be removed with the !Commands.Common.Clear_Underlining command.

Some of the units in the xref display may implicitly depend on a declaration but may not have direct usages of the declaration. Also, the Environment is conservative about finding all possible units that depend on the declaration. Sometimes it accidentally includes a unit in the xref that has no reference. In these cases, executing the Common.Definition command when designating such a unit deletes the unit from the xref and gives the message:

```
<unit name> doesn't have references!  Zapping the line.
```

Executing the Common.Semanticize procedure searches each unit in the xref and deletes any entries that have no usages.

procedure Common.Demote

Executes the Common.Demote procedure for Ada images on the selected unit.

procedure Common.Elide

Reduces the level of detail displayed in the current xref. This command is the opposite of the Expand command.

Although, by default, an xref displays the full name for each using unit, other levels of detail can be displayed using the Elide and Expand commands. These options are:

- Full_Names: Displays the full names of each unit with attributes (the default).
- Objects: Displays the unit name with attributes.
- Views: Displays the views using the declaration.
- Subsystems: Displays the subsystems using the declaration.

Executing the Elide command moves the level of detail down the above list; executing the Expand command moves the level of detail up the above list. The list is circular, so if you attempt to move down past the bottom, you go to the top; if you move up past the top, you go to the bottom. The current level of detail for an xref image is indicated in the banner for the xref.

procedure Common.Enclosing

Displays the library containing the unit for which the xref was created, with the unit selected.

procedure Common.Expand

Increases the level of detail displayed in the current xref. This command is the opposite of the Elide command.

Although, by default, an xref displays the full name for each using unit, other levels of detail can be displayed using the Expand and Elide commands. These options are:

- Full_Names: Displays the full names of each unit with attributes (the default).
- Objects: Displays the unit name with attributes.
- Views: Displays the views using the declaration.
- Subsystems: Displays the subsystems using the declaration.

Executing the Elide command moves the level of detail down the above list; executing the Expand command moves the level of detail up the above list. The list is circular, so if you attempt to move down past the bottom, you go to the top; if you move up past the top, you go to the bottom. The current level of detail for an xref image is indicated in the banner for the xref.

procedure Common.Explain

Displays the full name of the currently designated unit in the Message window.

procedure Common.Promote

Executes the Common.Promote procedure for the selected Ada images on the xref.

procedure Common.Release

Ends the editing of the xref image. The window is removed from the screen and from the Window Directory. This command has the same effect as the Abandon procedure.

procedure Common.Semanticise

Searches each unit in the xref for actual usages and deletes any entries for units with no usages.

procedure Common.Object.Child

Selects the unit on the line on which the cursor is located.

procedure Common.Object.First_Child

Selects the first unit in the xref.

procedure Common.Object.Last_Child

Selects the last unit in the xref.

procedure Common.Object.Next

Selects the next unit past the currently selected unit if the cursor is in the current selection; otherwise, the procedure selects the unit corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the unit corresponding to the line on which the cursor is located.

procedure Common.Object.Parent

Selects the unit corresponding to the line on which the cursor is located if there are no selections; otherwise, the procedure selects the entire list of units in the xref.

procedure Common.Object.Previous

Selects the previous unit before the currently selected unit if the cursor is in the current selection; otherwise, the procedure selects the unit corresponding to the line on which the cursor is located. If nothing is currently selected, the procedure selects the unit corresponding to the line on which the cursor is located.

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

L

M

N

O

P

R

S

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