Rational Environment Reference Manual

Debugging (DEB)

Document Control Number: 8001A-23

Rev. 2.0, January 1985
Rev. 3.0, October 1985
Rev. 4.0, December 1985
Rev. 5.0, July 1986
Rev. 6.0, 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 Debugging (DEB) book of the *Rational Environment Reference Manual* contains reference information describing the Rational Environment™ Debugger. This information is intended for users who are familiar with the Environment, Ada® programming, and the basic concepts of debugging programs using the Debugger. If you are not familiar with the basic concepts of using the Debugger, refer to the *Rational Environment User's Guide* and *Rational Environment Basic Operations* for a more user-oriented introduction to the Debugger.

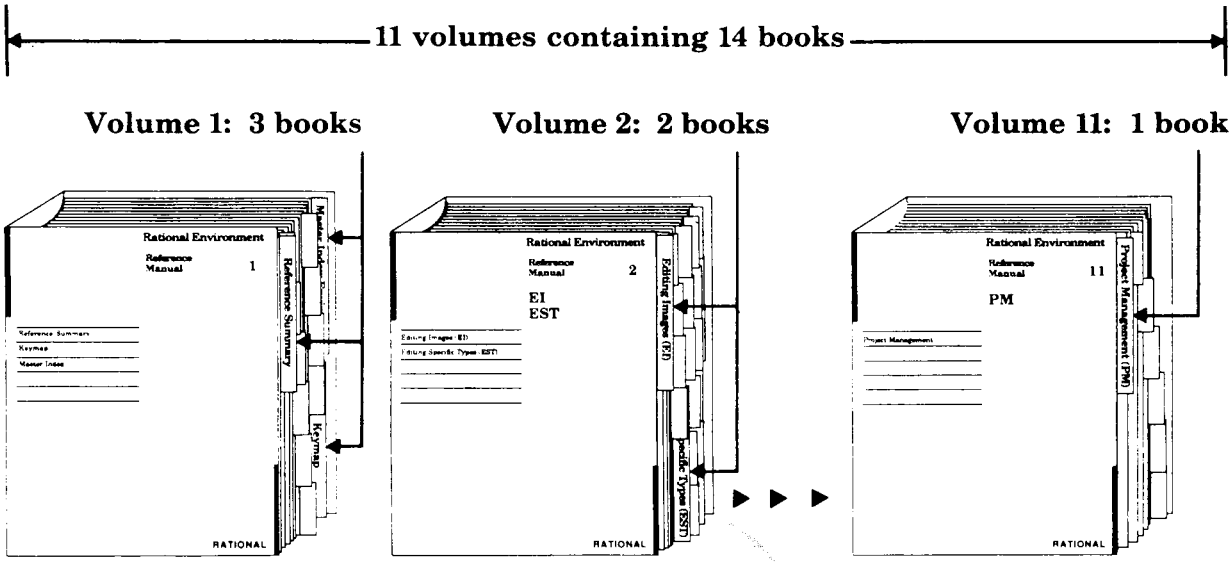
Organization of the Reference Manual

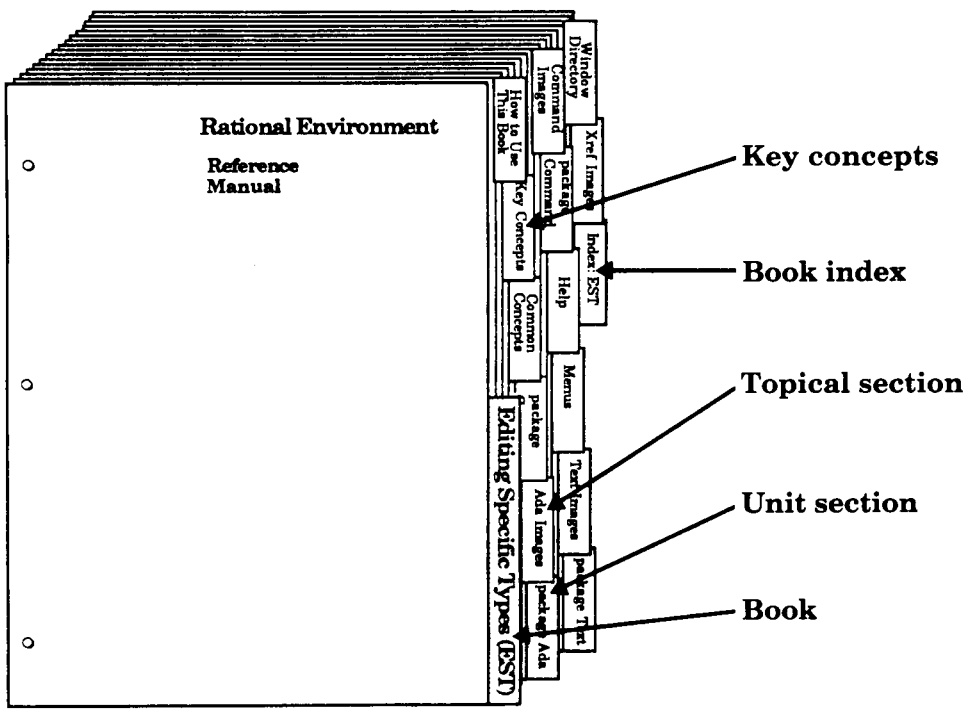
The *Rational Environment Reference Manual* (Reference Manual for brevity) includes the following volumes (see accompanying illustration):

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 Rational Environment Debugger provides a variety of facilities for analyzing the behavior of Ada programs running in the Rational Environment. The following reference information provides a detailed description of the Debugger and its operation. This information is intended for users who are familiar with the Environment, Ada programming, and the basic concepts of debugging programs using the Debugger. If you are not familiar with the basic concepts of using the Debugger, refer to the *Rational Environment User's Guide* and *Rational Environment Basic Operations* for a more user-oriented introduction to the Debugger.

Summary of Debugger Facilities

The Debugger provides facilities to:

- Display the source for any part of the program.
- Display the contents of the stack of any task in the program.
- Display and modify the values of variables in the program.
- Place breakpoints at various points within the program to trap the execution of portions of the program.
- Trace the execution of statements, subprogram calls, exceptions, and task interactions.
- Display tasks and their *state* (the current execution condition of the task) in the program.
- Execute specific tasks one statement, call, or rendezvous at a time.
- Stop execution when certain exceptions or groups of exceptions are raised.
- Control the execution of tasks in the program, stopping some and allowing others to continue.
- Record a history of the execution of various program events.
- Set a variety of parameters controlling the format and content of various Debugger displays.

Debugging Rational Environment Programs

A program in the Rational Environment is invoked by executing some number of statements and declarations in a Command window. The Environment creates a *job*, which executes the code in the Command window.

Each user of the Rational Environment can debug one job at a time. If debugging is in progress for one job and is then started for another job, the debugging process is automatically disabled for the first job and, by default, the first job is terminated.

Thus, the Debugger is always controlling one specific job. This job and the code that it executes are referred to as the *program* that is being debugged.

Programs can call facilities located anywhere in the Environment. The Debugger can operate on any such code that the program can execute. It does not matter where that code is declared.

To use the Debugger, no special options must be specified in the program. No special compilation is required. The Debugger is not compiled into your program but runs as a separate job, interacting with your program.

Tasks created by a job are part of that job. The Debugger is able to control only tasks that are part of the job it is currently debugging.

Debugging is enabled in a job when that job is executed with the !Commands-.Command.Debug procedure, which is normally bound to the Meta Promote key combination; see the Rational Environment Keymap, in Volume 1 of the *Rational Environment Reference Manual*, for all key bindings. The process of debugging a program is described in the sections that follow.

Debugger Interactions with the Editor

Interaction with the Debugger is through the Rational Editor mechanisms common to all operations in the Environment. Two packages in the Environment provide specific debugging services. Package !Commands.Debug defines interactive commands that initiate various Debugger actions and that produce output in the Debugger window. Package !Tools.Debug_Tools provides services that programs can use. It returns information in parameters or function results and usually does not result in output to the Debugger window.

Debugger Window

When the Debugger is first activated, the Environment creates a window called the *Debugger window.* All output from the Debugger appears in this window. It contains a complete log of all Debugger interactions for a session. The Debugger can also automatically display the Ada source for the program being debugged in an Ada window with the current location selected.

The Debugger window supports type-specific editing operations on it. Thus operations from package !Commands.Common apply to the Debugger window. Those

commands from package Common that apply to the Debugger window are described below. The common editing operations are discussed more fully in the documentation for package Common in EST, "Common Concepts and Operations." In addition to the commands in package Common, commands from other commands packages that take selections and so on can work using designations in the Debugger window. For example, the !Commands.Ada.Show_Usage procedure can be used when the item for which to find the usages is selected in the Debugger window, just as Show_Usage would be used from an Ada window.

Text in the Debugger window is not modifiable. The entire contents of the window can be written into a file by using the !Commands.Common.Write_File procedure. The contents of the Debugger window can be selectively copied (using the region copy operations) to other windows.

Each command executed is echoed in the Debugger window and followed with any output the command produces. If you want to execute the command again, you can get it to reappear in the !Commands.Command window from which it was initially executed by using the !Commands.Common.Undo procedure. For more information on that command, see EST, procedure Common.Undo. This command is also bound to a key in the standard Rational Environment Keymap.

If the Debugger window no longer appears on the screen (as a result of some other Environment operation), it can be redisplayed by pressing Debugger Window. The Debugger window automatically reappears if any Debugger command results in output being sent to it.

In the standard keymap, a number of common Debugger commands are bound to keys. These Debugger commands can be executed by pressing the appropriate key. See the Rational Environment Keymap for a list of Debugger commands bound to keys.

If the program you are debugging requests input from the terminal, you must execute the !Commands.Job.Interrupt procedure (the Control G key combination) after input has been committed to enter additional Debugger commands.

Automatic Source Display

The Debugger can display the current location in the program in an Ada window with the location highlighted when a breakpoint or step point is encountered. This facility is enabled by default. If you want to disable this facility, see the Debug.Disable command for the Debug.Option.Show_Location option.

Selection and Designation

Most Debugger commands accept the special names "<SELECTION>", "<REGION>", "<IMAGE>", and "<CURSOR>" to designate locations, objects, and so on. These designations can be performed in any Editor window, including the Debugger window.

The output from certain commands can be designated in the Debugger window. These commands and the specific output that can be designated are:

- Display: Any line of source in the Debug.Display command's output can be designated. Designating a line of output resolves to the object, statement, or declaration displayed on that line. Designation of a subprogram or package header line (for example, `procedure .DEBUGGING_EXAMPLE is`) resolves to the subprogram or package.
- Modify and Put: The command line echoed in the Debugger window can be designated. Designating this line resolves to the object that was put or modified. For example, if the output from the Debug.Put command is:

  ```
  Put ("%ROOT_TASK._1.A_VARIABLE");
  ```

  designating this line of output in the Debugger window resolves to the object called A_Variable in the first frame of the main program. Note that the designation resolves to a string name that the Debugger interprets as a pathname, not to the actual object. So if `"%ROOT_TASK"` or frame 1 has moved between the time the Put command echoed its output and the time the line of echoed output was designated, the new object is used.
- Stack: Any line of the frames displayed by the Debug.Stack command can be designated. Designating a line of output resolves to the statement or declaration corresponding to the designated frame. For example, designating the third frame of the following Stack command output resolves to the statement in procedure C that called procedure B in frame 2:

  ```
  Stack ("%ROOT_TASK", Ø, Ø);
  Stack of task ROOT_TASK, #1Ø74DE:
  _1: A.ls
  _2: B.ls
  _3: C.ls
  _4: D.ls
  _5: NESTED_CALLS.ls
  _6: command_procedure.ls
  _7: command_procedure     [library elaboration block]
  ```

See LM, Key Concepts, "Special Names," for more information on the available special names and how they are resolved.

Argument Prefixes

The argument prefix keys can be used to conveniently supply numeric values to Debugger commands bound to keys (these values can be stack frames, repeat counts, numeric flag values, and so on). The Debugger commands with only one integer-valued argument (there can be others of different types) accept arguments from the argument prefix keys. This facility, for example, can be used to run for five steps by pressing the argument prefix [5] and then [Run].

Session Switches

The initial values of various Debugger Boolean options, numeric value flags, and other flags that influence the behavior of the Debugger are read from the user's session switches when the Debugger is started. These session switches all have the form Debug_*xxx*, where *xxx* is the option, numeric value, or flag name.

See the reference entries for the Debug.Option type, Debug.Numeric type, and Debug.Flag procedure for more information on how these switch values influence the behavior of the Debugger. See SJM, Session Switches, for more information on session switches and the manipulation of them.

Commands from package Common

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

procedure Common.Abandon

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 below.

procedure Common.Create_Command

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

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

This *use* clause provides direct visibility to the declarations in packages 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.

procedure Common.Enclosing

Displays the library containing the Command window from which the job being debugged was started.

procedure Common.Release

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 above.

procedure Common.Write_File

Writes the current contents of the Debugger window into the named file.

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, it is selected again.

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 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.

Debugger Facilities

The Debugger provides a variety of facilities for the control and examination of Ada programs running in the Rational Environment. These facilities, and the commands involved for each, are described in the following sections. Full descriptions of all commands, parameters, and options can be found in the reference entries in packages Debug and Debug_Tools in this book.

The following reference information, which provides a detailed description of the Debugger and its operation, is intended for users who are familiar with the Environment, Ada programming, and the basic concepts of debugging programs using the Debugger. If you are not familiar with the basic concepts of using the Debugger, refer to the *Rational Environment User's Guide* and *Rational Environment Basic Operations* for a more user-oriented introduction to the Debugger.

Command Contexts

Many Debugger commands require a *context*. The context is additional information that specifies the task in the program to which the command should be applied or the manner in which names should be interpreted.

The default value for the task context is the last task stopped by the Debugger; the default for the location for name interpretation is the topmost subprogram activation (that is, the place where the task stopped). The last task and the topmost location are usually the task and location in which you are interested, so the defaults used by the Debugger usually refer to the location you want.

Most Debugger commands allow you to use the default value for the context or to name specifically a new context in the command. The Debugger also lets you set the contexts explicitly.

The *control context* specifies a task that will be the default value for commands and other operations requiring a task or stack. The evaluation context specifies a location relative to which unqualified Ada names given as parameters to Debugger commands will be interpreted.

For example, if the last task to stop, perhaps because of a breakpoint, is task %690E, the Stack procedure (with default parameters) displays the stack of task %690E. If you want to see the stack of task %137FB04, the command `Stack ("%137FBØ4")` displays it. Equivalently, if the control context is set to task %137FB04, the Stack procedure displays that stack by default.

As another example, the Display procedure, which displays the source code of the program, defaults to the top frame of the stack (_1) of the current task. The *current task* is the task specified by the control context if the control context is set; otherwise, it is the last task to stop in the Debugger. Thus, the Display procedure, by default, displays the source surrounding where the task stopped, giving the user a view of where the task is currently executing. If the evaluation context is set explicitly, the location it specifies is displayed by the Display procedure. The Display procedure also takes an argument that specifies what to display.

The current contexts can be displayed by using the command `Show (Contexts)`.

Tasks, Task State, and Task Control

Each program consists of one or more tasks. The task that is the *main program* is called the *root task*. All other tasks in the program are declared or allocated by the root task or by tasks declared or allocated by the root task.

To enable debugging for an entire program, the program must be executed with the !Commands.Command.Debug procedure. This is normally done with the Meta Promote key combination. If a program is being debugged when the Debug procedure is issued, that program is killed.

Each task in the program is assigned a number by the Environment. The number can be used to refer to the task in various debugging operations.

Individual tasks can also assign themselves *string names.* They do this by calling the Debug.Set_Task_Name procedure or the Debug_Tools.Set_Task_Name procedure with a string name parameter. The root task, by default, is named "Root_Task." That string name can always be used for it. It is good programming practice to set string names for each task, especially if the same task type occurs in many instances.

The Debugger displays the string name along with the task number whenever it displays information about the task. The string name can also be used in place of the task number in parameters of commands that specify a task.

The Debug.Task_Display procedure lists all or a subset of the tasks in the current program; it lists the number and string name (if any), the Ada name, and the state of each task. The Task_Display procedure can list subsets of all tasks, such as those that are actually running, blocked, or stopped by the Debugger.

Call Stacks

Execution of block-structured, procedural languages such as Ada involves the stacking of activations of subprograms and blocks. When a subprogram is called, a *stack frame* (or, simply, a *frame*) is pushed on the stack of the task executing the call. The frame contains the values of local variables and parameters to that subprogram.

Information available from each frame includes the values of local objects (such as variables, packages, and tasks) declared in that frame, parameters to that call, and the name of the subprogram that is executing with that frame as its context. Because subprogram calls can be recursive and because different tasks can execute the same subprogram, it is necessary to specify a task and a specific frame in order to examine a specific local variable or parameter of a subprogram.

Blocks and accept statement bodies are treated as independent subprograms; consequently, they get their own frames.

The Stack procedure displays the stack of a specific task.

Stopping and Holding Tasks

Individual tasks can be stopped or allowed to continue execution by the Debugger. Note that if many tasks occur in a program, those tasks cannot be stopped or allowed to proceed simultaneously (that is, as an atomic operation). The user can affect the program's behavior by using the Debugger to stop and start tasks. The tasks always behave consistently with Ada semantics, however.

The Debugger can also be set up to stop all tasks when any task stops in the Debugger, allowing debugging of a tasking program in a more single-thread manner. Note that, in this mode, other tasks may not actually stop because they may be in rendezvous with the stopped tasks, waiting for an entry call, and so on. When the task that stopped is restarted, any tasks stopped because of the Freeze_Tasks option are restarted. To enable this mode of operation, the Freeze_Tasks option must be set to true (its default value is false). See the description of the Debug.Enable procedure for more information on setting this flag.

Task execution may be interrupted by the Debugger in a number of ways, including the use of breakpoints, exception handling, stepping, and other Debugger operations. In this section, only the Debugger's facility to stop and hold tasks is discussed. The other facilities are discussed in the following material.

The Debug.Stop, Hold, Release, and Execute procedures provide the ability to control groups of tasks and keep some of them inactive while allowing others to execute and interact. They allow individual or all tasks to be stopped and then to continue.

From the Debugger's point of view, a task is in one of three states: ***running***, ***stopped***, or ***held***. In the running state, the task is free to execute normally. The task need not be executing; it may be blocked (waiting for an entry call to be accepted, for example). When the Stop procedure is issued, the task moves into the stopped state when it completes the currently executing statement. In this state, the task does not execute.

The Execute procedure causes tasks in the stopped state to continue their execution (changing their state to running). The Execute procedure can be applied to either an individual task (leaving any others in the stopped state) or to all tasks in the stopped state, starting them all.

To debug a few interacting tasks and to keep others from executing, the Hold procedure can be used. The Hold procedure works like the Stop procedure except that the task goes into the held state. Tasks in the held state are not started by the command `Execute ("all")`. The task can be started by an Execute procedure that names it explicitly. The task can be moved from the held to the stopped state by the Release procedure. Once the task is in the stopped state, executing either command `Execute` ***(task name)*** or command `Execute ("all")` starts the task.

Using the Hold procedure, you can remove a few specific tasks from the set of tasks that are running and stopping, thereby debugging only the nonheld tasks. You can also use the command `Hold ("all")` and the Release procedure to release a few specific tasks, keeping most tasks held and debugging only a few.

The command `Show (Stops_And_Holds)` displays the tasks that currently have stops and holds applied to them.

Other Information on Task State

The Debugger provides a few miscellaneous facilities that can be used to examine task state.

The command `Information (Exceptions)` displays the names of any exceptions that are currently being handled by a specific task and the location in which the exception was raised.

The command `Information (Rendezvous)` displays the name of the task with which a specified task is rendezvoused. This command is useful in tracing task interactions and parameters back across entry calls.

The command `Information (Space)` displays the names of each task in the program and the amount of control and data space each task is using as well as the maximum data space it has used.

Breakpoints

A *breakpoint* is one of the mechanisms that allow the user to control the execution of a task. While the task is stopped at a breakpoint, objects can be viewed or modified, source can be displayed, and breakpoints, traces, histories, or exception handling can be added, changed, or deleted. Other tasks can also be stopped or held. Execution can then be restarted.

Execution of an Ada program involves the *elaboration* of declarations followed by the *execution* of statements. Unlike most other languages, the elaboration of declarations can cause execution of other subprograms (from a call in the initialization of a variable, for example) and can also cause exceptions to be raised.

The Debugger allows breakpoints to be set at declarations as well as at statements. Such a setting allows you to debug actions that occur during the elaboration of declarations.

To reference statements and declarations in the program, declarations and statements in each subprogram are numbered by the Debugger. Declarations and statements are grouped separately and numbered sequentially starting with 1. A number is assigned at the beginning of each declaration or statement. Blocks and accept bodies are also numbered starting with 1. The naming of locations is discussed in detail under the Path_Name subtype in package Debug in this book.

The Display procedure shows the numbers of declarations and statements as part of displaying the source of a subprogram.

Assume the following simple subprogram is a library unit:

```
Display (".swap", Ø);
        procedure .SWAP  (X, Y : in out INTEGER) is
    1       TEMP : INTEGER;
        begin
    1       TEMP := X;
    2       X := Y;
    3       Y := TEMP;
        end;
```

Breakpoints can be set at declaration 1 and statement 2 by issuing the commands:

```
Break (".Swap.1d");

Break (".Swap.2");
```

Note that although the pathnames of the locations to set these breakpoints were provided explicitly, breakpoint locations typically are specified with selections so that the names of the locations do not have to be typed in.

These breakpoints stop any task in the program being debugged that executes the Swap procedure. The location and task name are reported in the Debugger window.

Breakpoints can be set so that only a specific task will stop when it reaches the specified location. This setting allows debugging of individual tasks, not just the code that any or all tasks execute. The In_Task parameter to the Debug.Break procedure can specify the task to which the breakpoint should apply. The Break procedure defaults to set the breakpoint in the task specified by the control context. If the control context is not set to a specific task, the breakpoint applies to all tasks.

The Break procedure has several other parameters. In addition to a restriction to a specific task, a breakpoint has a ***trip count*** associated with it. The trip count specifies the number of times that the location where the break is set must be executed before the Debugger stops the task that executes there. The trip count is useful for breakpoints that are set in loops.

A breakpoint can be *active* or *inactive*. When a breakpoint is active, it is installed in the program being debugged and stops a task that executes the location where it is placed (when any task and trip count restrictions are satisfied). When a breakpoint is inactive, it has no effect on the execution of the program. The Debugger simply remembers all the information about the breakpoint so that the breakpoint can be reactivated.

Active breakpoints are deactivated when a new job is started with debugging. The command `Activate (0)` reactivates all breakpoints. This command can be used when a program is being rerun and you want to break at the same locations as the last run.

The Break procedure creates and activates a breakpoint. The Activate procedure reactivates an inactive breakpoint. The Remove procedure deactivates or deletes a breakpoint, or both. Each breakpoint is assigned a number by the Break procedure. This number is used to refer to the breakpoint in subsequent Debugger operations. The number is also used when the Debugger reports that a task has stopped at a breakpoint.

Breakpoints can also be designated as either *temporary* or *permanent*. When a task is stopped by a temporary breakpoint, the breakpoint is automatically deactivated (and optionally deleted). If you are advancing through a program and do not want to retain breakpoints as they are passed, creating them as temporary is useful. A temporary breakpoint causes one task, at most, to stop. Permanent breakpoints remain active until explicitly removed. The Default_Lifetime parameter to the Break procedure can be used to set either permanent or temporary breakpoints. See the reference entry for the Break procedure for more information on the use of this parameter.

Tracing

The *tracing facility* of the Debugger causes messages to be displayed each time a certain kind of event occurs in a task for which tracing is enabled.

Traces can include messages about statements, calls, rendezvous, and exception raising. Traces are displayed in the Debugger window or recorded in a file (see the Debug.Trace_To_File procedure.

Tracing can be enabled for specific tasks or for all tasks. The Trace procedure is used to enable and disable tracing. The command `Show (Traces)` displays what tracing is enabled for which tasks. If tracing is enabled for one task, other tasks are not affected (except that the task with tracing is running more slowly than normal).

History

The *history facility* of the Debugger causes messages to be placed in a buffer each time a certain kind of event occurs in a task for which history taking is enabled.

As with traces, histories can include messages about statements, calls, rendezvous, and exception raising. Unlike traces, history messages are saved in a circular buffer in the Debugger. From the buffer, selected sets of messages can be displayed, such as messages from only a specific task or for some range of messages.

Like traces, histories can be enabled for specific tasks or for all tasks. The Debug.Take_History procedure is used to enable and disable histories. The command `Show (Histories)` displays what tracing is enabled for which tasks. If history taking is enabled for one task, other tasks are not affected (except that the task with history taking is running more slowly than normal). The History_Display procedure displays the actual history information.

Exception Trapping

Exceptions raised in a program often signal some event of interest during debugging. The Debugger provides a facility to stop execution of a task when it raises particular exceptions. When this happens, the Debugger reports the stopping of the task, the name of the exception that was raised, and the location in the program where it was raised.

In a given program, some exceptions may or may not be of interest to you during debugging. You may care about exceptions only when they are raised by specific tasks or in specific locations.

The Debugger maintains a list of exception information requests. A request may ask that a task stop for a certain exception (a *catch request*) or that a task not stop for a certain exception (a *propagate request*).

The Debug.Catch, Propagate, and Forget procedures add catch, propagate, and delete requests, respectively.

For example, if you want the program you are debugging to stop when any exception other than the Constraint_Error exception is raised, use the following commands:

```
Propagate ("Constraint_Error");
Catch;
```

After an exception is caught by the Debugger and reported by a message in the Debugger window, and after the task in which the exception is raised is stopped, you can inspect the state of the task (or other tasks). You can allow the task to continue when you wish. Execution continues with the first statement of the handler for the exception that was raised.

If a task stops at a point inside an active handler for an exception (such as at a raise statement), the command `Information (Exceptions)` displays information about the exception, including where it was initially raised.

The Catch, Propagate, and Forget procedures take additional parameters that restrict exception requests to specific tasks or locations. Because overlapping requests are possible, rules specify which requests apply in a given situation. The additional parameters and rules are discussed under the Catch and Propagate procedures in package Debug in this book.

If the Save_Exceptions option is set to true (its default value is false), the exception-handling information is saved between debugging runs. This mode can be used when a program is being rerun and you want to handle exceptions in the same way as the last run. See the description of the Debug.Enable procedure for more information on setting this option.

The command `Show (Exceptions)` lists all catch and propagate requests in the order in which they will be applied to each exception situation.

By default, the Debugger catches all exceptions in all tasks in the program wherever they are raised. This response to exceptions is the effect of the command `Catch` (with default values for all parameters). The command `Propagate` (again, with default values for all parameters) changes this default to indicate that tasks are not to stop when exceptions are raised (if no other catch requests are entered).

Stepping

The Debugger provides operations for executing tasks one statement, call, or task interaction at a time. The command `Run` (with default values for all parameters) causes a task that is stopped in the Debugger to execute one statement.

Execution continues until the specified number of the specified kinds of events occur. For example, the commands:

```
Run (Statement, 5);
Run (Local_Statement, 2);
Run (Procedure_Entry);
Run (Returned);
```

run the current task for the next five statements, for the next two statements of the current subprogram, until the beginning of the next called subprogram, and until the first statement that follows the call to the current subprogram, respectively.

Note that the Statement parameter specifies stepping after any statement. The Local_Statement parameter counts only statements in the current subprogram (and the current subprogram's caller) and treats called subprograms and all statements contained therein as one statement.

While it is stepping, a task may stop for some other reason. For example, it may encounter a breakpoint or may raise an exception that the Debugger is requested to catch. When the task is resumed, stepping continues and the task stops when the stepping request has been fulfilled.

The command `Show (Steps)` displays the names of tasks that are currently stepping.

An in-progress stepping operation can be canceled by executing the Debug.Clear_Stepping procedure. This command causes the stepping operation to be canceled, and the task continues doing whatever it was doing. Other Debugger operations applied to the task are unaffected.

Displaying and Modifying Program Data

The Debug.Put and Modify procedures provide the capability to display and modify data in the program being debugged. The Put procedure is given the name of the object to be displayed. If the object is a scalar, the simple value is displayed. If the object is a structure, the entire record or (part of) the array is displayed. If the object is a pointer, then the pointer value is displayed followed by the object that it designates.

The Debugger provides a facility for allowing users to create special display routines for displaying data values. If the data being displayed is of a type not defined in the application (for example, a private type defined in the one of the Ada specifications for the Rational Environment) and consequently not known directly by the Debugger, it can also be displayed with this facility. Users must provide display functions for the type and register them with the Debugger (typically in the Debugger_Initialization procedure; see below). These functions can be created and registered using the Debug_Tools.Register generic procedure. See the reference entry for this procedure in this book for more information.

Normal Ada notation for record fields and array subscripts is used when specifying objects to be displayed. Thus, individual fields of structures can be named, as can the designated objects of pointers.

Scalar-valued variables can be modified by using the Modify procedure. This command also allows scalar components of records and arrays to be modified. Some restrictions limit what can be modified. See the Debug.Modify procedure in this book for a list of restrictions.

If the Display procedure is given a data object or type name, it displays the structure of the object's type. This display is very useful in conjunction with the Put procedure to see the declaration of the type using the Display procedure and the display of part or all of the object using the Put procedure.

The command `Set_Value (Display_Level, depth)` specifies the number of levels of nesting that are displayed when a structured value is displayed. Other options (First_Element and Element_Count) control the display of arrays (see Numeric type).

If a number of Put, Modify, or Display procedures are to be executed for similar objects, the evaluation context can be set to simplify the entry of the operations. The evaluation context is set to the run-time context that contains the objects of interest. Then, simpler object names (relative to the context) can be used to name the objects.

For example, suppose task %4112 has stopped at a breakpoint. The subprogram active in stack frame 2 contains a package, Storage_Map, that in turn contains a pointer, Active_Data, to a record. You may want to display the value of a field of that record, Motor_Status. You can display this value by using the command:

```
Put ("%4112._2.Storage_Map.Active_Data.Motor_Status");
```

If several fields (for example, Engine_Status) or subfields are to be displayed, then the evaluation context can be set:

```
Context (Evaluation, "%4112._2.Storage_Map.Active_Data");

Put ("Motor_Status");

Put ("Engine_Status");

Put ("Engine_Status.Temperatures.Oil.Filter_2");
```

You must be aware that the evaluation context will have this value until explicitly changed. All unqualified names will be interpreted relative to it. The command `Context (Evaluation, "")` clears the context so that it will again default to the top frame of the last task to stop (or the task specified by the control context, if that is set to some task). The command `Display (Contexts)` displays the current value of the evaluation and command contexts.

If the task had not been stopped, the Debugger could still display the value of the field. However, the value the Debugger displays could change at any time because the task is still running. The Debugger displays the message:

```
Warning: Task #4112 is running
```

along with the value displayed to remind you that the value may change.

Miscellaneous Facilities

A few details not covered in the preceding sections are worth noting.

Debugger_Initialization Procedure

To perform initialization of the Debugger when it is started, put a parameterless procedure called Debugger_Initialization in a library that is on your searchlist. This procedure is elaborated and run at the point the Debugger is started. It remains elaborated until the Debugger is killed. This facility can be useful in setting up flags and exception handling and in registering special display procedures for specific types (see the Debug_Tools.Register procedure for more information on such display procedures).

Starting the Debugger Quietly

To automatically start the Debugger in your login procedure so that you do not have to wait for it to start when you debug the first program in your session, you can do a quiet startup. Calling the Debug.Reset_Defaults procedure from your login causes the Debugger to start without bringing up the Debugger window.

Numeric Conversion

The Debug.Convert procedure converts numbers from one base to another, most notably hexadecimal and decimal.

Options, Numeric Flags, and Flags

A number of Boolean options, numeric flags, and flags control various aspects of the Debugger's operation. These options are set and reset with the Debug.Enable and Debug.Disable procedures (for Boolean-valued flags), the Debug.Set_Value procedure (for numeric-valued flags), and the Debug.Flag procedure (for any other flags). See the reference entries for the Debug.Option and Debug.Numeric types and the description of the Debug.Flag procedure for detailed information on all controllable aspects of Debugger behavior.

Programmatic Access to Debugger Facilities

The Debugger provides several operations in package !Tools.Debug_Tools that are callable directly from tasks that are part of an Ada program.

A running task can call the Debugger explicitly to display a message or to "break" to the Debugger. Displaying a message is useful because the program can inform you by means of the Debugger that it has reached a certain point or has detected some abnormal condition.

When desired, a call to the User_Break procedure stops execution of the calling task as though it had encountered a breakpoint. A message is displayed in the Debugger window indicating the name of the task. The Execute procedure causes the task to continue; from the task's point of view, the User_Break procedure returns.

Calls to the User_Break procedure have no effect from a job that is not currently being debugged.

Three operations provide a running program with information about itself: the Ada_Location, Get_Exception_Name, and Get_Raise_Location functions.

The Ada_Location function allows a task to scan its stack and get information about what is executing in each frame. This scan is useful for displaying error information in certain cases, because it shows the exact state of the program at the time the error is detected.

The Get_Exception_Name function, when called directly or indirectly from an exception handler, returns in string form the name of the exception that was raised. Its raise location can be displayed with the Get_Raise_Location function.

Debugger Naming

Pathnames Referencing Ada Programs

Many operations require that specific locations in the program be named. These locations can be any statement, declaration, exception, object, or type. Strings, called *pathnames*, are used to specify these locations. These names can specify locations as simple as an object to as complex as a specific statement in a specific generic instantiation. These same names are also used by the Debugger when it displays locations where a program task has stopped. Note that pathnames to Debugger commands accept special names (such as "<SELECTION>"). See "Selection and Designation," above, for more information on the use of special names and designations in the Debugger window. See LM, Key Concepts, "Special Names," for more information on special names and their resolution.

Pathnames generally use existing Ada naming rules for program units. Special cases in which Ada rules are extended include names for anonymous blocks, package specs and bodies, accept statement bodies, locations within generic program units, and statements and declarations. Each instance of a generic, as well as the generic itself, has a unique pathname.

For the purposes of naming, statements and declarations are numbered. Numbering within each group of declarations and statements is independent and begins with 1. *Use* clauses and representation specifications are not numbered. Blocks and accept statement bodies are numbered separately. Statements in exception handlers are numbered continuously with preceding statements.

The pathname of a statement or declaration includes the pathname of the containing subprogram, package, task, block, or accept, and a statement or declaration number suffix.

Names of data objects follow Ada rules for the selection of record fields, array elements, and objects designated by pointers. Even if a type is declared private, the Debugger allows you to access it as though the structure were known, unless it is a Rational Environment private type.

Pathnames are always interpreted in a left-to-right sequence. At each point during name processing, the processed left part of the name has been resolved to some Ada

unit, and the next name component is interpreted within that context. The same interpretation method is used for Environment names.

Pathnames are used in most Debugger commands. The Display procedure, for example, takes a pathname that specifies what source is to be displayed. The Put procedure takes a pathname that specifies a data object whose value is to be displayed. The Break procedure takes a pathname that specifies a location in the program where a breakpoint is to be placed. Other commands also make use of pathnames to identify program or data locations.

Special Characters in Names

Special characters are used in names to specify either relative or absolute contexts. These special characters apply to names used throughout the Environment, not just in the Debugger.

A special character in a name determines the context in which the remaining portion of the name will be interpreted. A special character such as an exclamation mark (!), caret (^), dollar sign ($), double dollar sign ($$), percent symbol (%), period (.), or underscore (_) causes an explicit interpretation of the remainder of the name as described below. Some of these characters can be used only at the beginning of the name. Note that the < character is also special in the sense that it indicates a special name (for example, "<SELECTION>") is being used.

If the first character of the name is not one of these characters, then the name is said to be an *unqualified name*, and the Debugger uses the control and evaluation contexts as defaults in which to interpret the name (also described below). These defaults are chosen so that the interpretation will likely be in the context in which you are interested.

The Special Character !

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

The Special Character ^

The caret (^) specifies that the context should be set to the immediately enclosing object from the location in which the Debugger was executed. This climbs the hierarchy of objects and eventually reaches the root of the library system. The parent object of the root of the library system is itself.

The Special Character $

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

The Special Character $$

The double dollar sign ($$) specifies that the context should be set to the immediately enclosing world from the location in which the Debugger was executed. If the current context is a world, this character has no effect.

The Special Character %

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

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

The Special Character .

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

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

The Special Character _

If the value of an object declared in a subprogram is to be displayed, then the frame on the run-time stack that contains an activation of that subprogram must be named. Naming is done using the notation *_frame number*. Stack frames are numbered for each task starting at the top with 1. Thus, _4 refers to frame number 4 (fourth frame from the top). (Frames are alternately numbered from the bottom using negative numbers.)

Note: In the Debugger and throughout the Environment, the convention is that the top of the stack is the most recently used location.

Task and frame names can be combined to specify a frame of a specific task; for example, %32912._4 specifies frame number 4 of task number 32912.

If a name begins with a frame specification (_*n*), the task referred to is based on the value of the control context. If the control context is set to some specific task, the frame reference refers to this task's stack. If the control context is not set to a specific task, the frame reference refers to the stack of the last task to stop in the Debugger. This task is usually the one in which you are interested.

Other Special Characters

Two additional characters that can be used in Debugger names are:

\ Indicates that the next name component is resolved in the current context and the current searchlist.

` Indicates that the next name component is resolved in the current context and with the current links using Ada naming rules.

Unqualified Names

Finally, if a name does not begin with an exclamation mark, caret, dollar sign, double dollar sign, percent symbol, period, or underscore, then the name is said to be an ***unqualified name.*** Unqualified names are interpreted in the Debugger's evaluation context as if the evaluation context were prepended to the pathname (with appropriate connecting punctuation). If the evaluation context is not set (that is, if it is set to the null string, its default value), then _1 is prepended to an unqualified pathname. The name is then interpreted as an object declared in the top frame of the task specified by the control context or in the last task to stop if the control context is not set.

Referencing Library Units

All library units referenced in a program are known to the Debugger. Some important points about how to name those units follow. Consider the following library structure and the program in it:

```
!Users.Airport.Airport_System  :   Library (World);
  Main_Program : C Ada (Proc_Spec);
  Main_Program : C Ada (Proc_Body);
  Source       :   Library (Directory);
  Tests        :   Library (Directory);

!Users.Airport.Airport_System.Source  :   Library (Directory);
  Some_Other_Unit : C Ada (Proc_Spec);
  Some_Other_Unit : C Ada (Proc_Body);
  ...
  Some_Unit       : C Ada (Proc_Spec);
  Some_Unit       : C Ada (Proc_Body);

!Users.Airport.Airport_System.Source  :   Library (Directory);
  Some_Unit    : C Ada (Proc_Spec);
  Some_Unit    : C Ada (Proc_Body);
  ...
  Test_Data    :   File;
  Test_Driver  : C Ada (Proc_Spec);
  Test_Driver  : C Ada (Proc_Body);
```

Assume that the program Main_Program imports library units via ***with*** clauses from both the directory Source and the directory Tests. The units from within the world Airport_System can be directly imported into the main program via internal links in the world.

The Debugger allows references to these library units with pathnames such as .Some_Unit.2d or .Test_Driver.Some_Subprogram. These pathnames do not need to be qualified with the library name. Thus the Debugger "flattens" the library structure from which the program was built.

The program might also want to use some facility in the Environment such as Text_Io. In the world Airport_System, there would be an external link to the library unit Text_Io. The Debugger also allows using pathnames that directly reference that imported library unit, such as .Text_Io.Put_Line.

Because the Debugger flattens the library structure, it is possible that library unit names will become overloaded from the point of view of the Debugger. This could lead the Debugger to not know to which unit, called Some_Unit, reference is being made. Three resolutions are possible:

- An ambiguous reference to a library unit is handled just as are other overload resolutions in the Debugger (see "References to Overloaded Subprograms," below).
- The containing library name can be used to specify to which unit reference is being made, such as .Tests.Some_Unit or .Source.Some_Unit.
- Fully qualified names can be used.

Referencing Data Structures

Pathnames can refer to Ada variables. A pathname can refer to a specific field of a record, a component of an array, or an object that is pointed to by an access value.

Pathnames always start in some part of the program, such as a package, task, or subprogram frame, or they start in a library. From there, the pathname can follow a path that includes data values.

Consider the following example. Assume that the world Airport_System is in the library !Users.Airport; package Reservation is a library unit therein.

```
      package RESERVATION is
1       subtype PASSENGER_NAME is STRING (1 .. 20);
2       type FLIGHT_INFO is ...
3       type PERSON;
4       type PERSON_PTR is access PERSON;
5       type PERSON is record
            NAME : PASSENGER_NAME;
            AGE : NATURAL;
            FLIGHT : FLIGHT_INFO;
            NEXT : PERSON_PTR;
      end record;
6     procedure RESERVE (PASSENGER_LIST : in out PERSON_PTR;
                         NAME           : PASSENGER_NAME;
                         AGE            : NATURAL;
                         FLIGHT         : FLIGHT_INFO);
      end RESERVATION;
```

Suppose that the last task to stop in the Debugger is executing in the Reserve procedure of the above example, and that neither control nor evaluation contexts are set. Then the name Passenger_List.Next.Next.Name(12) refers to a character at index 12 of an array (string) two links down a linked list of records. Note that, as in Ada, the access variable Passenger_List is automatically dereferenced. The name Passenger_List.All refers to the record designated by the variable Passenger_List.

If the activation of the Reserve procedure were in a task with name Reservation_Agent at frame 7, then the name:

```
%Reservation_Agent._7.Passenger_List.Next.Next.Name(12)
```

would reference the same character as the first example.

The name:

```
!Users.Airport.Airport_System.Reservation.Reserve.Passenger_List.Flight
```

follows a pathname from the root of the Environment to a field of a record. Note that no run-time value is associated with the object to which this name refers; a subprogram is referenced, but a specific activation (frame) of that subprogram is not referenced. Such a name could be used to display information about the type of the object to which it refers but not about a run-time value.

The name:

```
.Reservation.Reserve.Passenger_List.Flight
```

has the same meaning as the previous example.

Referencing Programs

Using name constructs discussed so far, it is possible to name individual subprograms, packages, and tasks in an Ada program. To refer to individual statements, declarations, blocks, and certain task constructs, the naming rules are extended.

Within each subprogram, package, task, block, and accept body, statements and declarations are numbered beginning with 1. In the display of the source provided by the Debugger, the numbers are shown.

To name a specific statement or declaration, the name of the containing subprogram is suffixed with the statement or declaration number. Declaration numbers are suffixed with `d` to distinguish them from statement numbers. Statement numbers can optionally be suffixed with `s`.

Blocks are treated as independent subprograms and are referenced as a whole by the statement number of the block in the subprogram that contains it. If the block is labeled, then the label can be used instead of the statement number. See the examples in the following sections for more details.

Accept statement bodies are treated similarly to blocks. Statements within an accept body are numbered independently, and the accept statement body itself is identified by the number of the accept statement in the statement list that contains it.

Select alternatives are numbered sequentially with surrounding statements. See the examples in the following sections.

Referencing Overloaded Subprograms

When a reference is made to a subprogram name and that name is overloaded, the Debugger requires an ***overload resolution nickname*** to make the subprogram reference unambiguous. An overload resolution nickname is a number or identifier that is appended to the subprogram name as an attribute. The number is chosen by the Environment or a name can be set by the user. If a name references an overloaded subprogram and does not include the overload resolution nickname, then the Debugger displays each of the alternatives along with its number, as in the following example:

```
Display ("%ROOT_TASK._1.put", Ø);
 The name PUT is overloaded.
 When you ask for it again, please choose one instance by appending
 to PUT a single quote followed by a choice qualifier or nickname,
 as follows:

 Choice N(1):
       procedure .OVERLOADS.PUT'N(1)   (I1, I2 : INTEGER) is

 Choice N(2):
       procedure .OVERLOADS.PUT'N(2)   (I : INTEGER) is

 Choice N(3):
       procedure .OVERLOADS.PUT'N(3)   (S : STRING) is

Invalid location specified:
Unresolvable overload: PUT
```

If the Put procedure that takes the single integer parameter is desired, its overload nickname can be included:

```
Display ("%ROOT_TASK._1.put'n(2)", Ø);

   procedure .OVERLOADS.PUT'N(2)   (I : INTEGER) is
   begin
1     TEXT_IO.PUT (I);
   end;
```

This overload resolution process is also used when a specification might be interpreted either as the visible part or body of a package. For example, given the package:

```
   package .OVERLOADS.FOO_BAR  is
1      A : INTEGER;
   end;

   package body .OVERLOADS.FOO_BAR   is
1      B : INTEGER;
   end;
```

a request to display package Foo_Bar declaration 1 would be ambiguous:

```
Display ("foo_bar.1d", 0);
 The name 1D is overloaded.
 When you ask for it again, please choose one instance by appending
 to 1D a single quote followed by a choice qualifier or nickname,
 as follows:

 Choice N(1):
       A : INTEGER;

 Choice N(2):
       B : INTEGER;

Invalid location specified:
Unresolvable overload: 1D
```

Then, the name foo_bar.1d'n(1) refers to the declaration of A, which is in the visible part, and foo_bar.1d'n(2) refers to the declaration of B, which is in the body.

It is also possible to refer specifically to the visible part or body of a package by appending the nickname "'spec" or "'body" to the package name, respectively. For example, foo_bar'spec.1d always refers to the declaration (A) in the visible part.

Referencing Generic Instantiations

Code within generic program units can be referenced in two ways: by naming the generic unit itself and by naming an instance of the generic unit.

When breakpoints are set, the two names have different effects. If the generic unit is named, the breakpoint applies to all instances of the generic. If an instance is named, the breakpoint applies only to that one instance of the generic.

Several examples involving generics are given in the following material.

Naming Example

Again, assume that the following package is a library unit in the library !Users-.Airport.Airport_System:

```
        package RESERVATION is
 1      subtype PASSENGER_NAME is STRING (1 .. 20);
 2      type FLIGHT_INFO is ...
 3      type PERSON;
 4      type PERSON_PTR is access PERSON;
 5      type PERSON is record
            NAME   : PASSENGER_NAME;
            AGE    : NATURAL;
            FLIGHT : FLIGHT_INFO;
            NEXT   : PERSON_PTR;
        end record;
 6      procedure RESERVE (PASSENGER_LIST : in out PERSON_PTR;
                           NAME           : PASSENGER_NAME;
                           AGE            : NATURAL;
                           FLIGHT         : FLIGHT_INFO);
        end RESERVATION;

    package body RESERVATION is
```

```
1  procedure RESERVE (PASSENGER_LIST  : in out PERSON_PTR;
                      NAME            : PASSENGER_NAME;
                      AGE             : NATURAL;
                      FLIGHT          : FLIGHT_INFO) is
   begin
   1    CHECK_NAME (NAME);
   2    ADD_TO_FLIGHT (PASSENGER_LIST, NAME, FLIGHT);
   3    ADD_TO_LIST (PASSENGER_LIST, NAME, AGE, FLIGHT);
   4    declare
      1     COST    : TICKET_PRICE;
        begin
      1     COST := COMPUTE_TICKET_PRICE (FLIGHT);
      2     SEND_BILL (NAME, COST);
        exception
           when others =>
      3        CANCEL_RESERVATION (NAME, FLIGHT,
                                    PASSENGER_LIST);
        end;
   5    FINALIZE_TRANSACTION;
   end RESERVE;
end RESERVATION;
```

First, note that each declaration and statement is numbered. These numbers are used to name specific declarations and statements. Consider the following examples from the code segment above:

1. !Users.Airport.Airport_System.Reservation'spec.2d
2. !Users.Airport.Airport_System.Reservation'body.1d
3. !Users.Airport.Airport_System.Reservation.Reserve.2
4. !Users.Airport.Airport_System.Reservation.Reserve.4.1d
5. !Users.Airport.Airport_System.Reservation.Reserve.4.3
6. !Users.Airport.Airport_System.Reservation.Reserve.5

Examples 1 and 2 refer to declarations in the visible part and body of package Reservation, respectively. Examples 3 and 6 refer to numbered statements in the Reserve procedure. Examples 4 and 5 refer to the declaration for Cost and the statements in the exception handler of the block within Reserve.

Another Naming Example

An example of statement and declaration numbering follows:

```
      with BOUNDED, TTY_IO;
      procedure TEST8 is
          use BOUNDED, TTY_IO;
1         L : BOUNDED.VARIABLE_STRING(500);
2         task type USER ...;
3         type R2 is record
              F1 : INTEGER := 21;
              F2 : CHARACTER := '2';
          end record;
4         type R1 is record
              F1 : INTEGER := 1;
              F2 : INTEGER := 2;
```

```
        end record;
5       DONE : exception;
6       procedure FOO_PROP (N : INTEGER) is ...;
7       V1 : R1;
8       V2 : R2;
9       procedure REC_TEST (V : R1; U : in out R2) is ...;
    begin
1       RANDOM.INIT_RANDOM(5);
2       PUT ("test8 starting");
3       NL;
4       REC_TEST (V1, V2);
5       GTEST_MAIN;
6       begin
    1       FOO_PROP (5);
        exception
            when others =>
    2           PUT ("Back from Foo_Prop");
        end;
7       SESSION.INITIALIZE;
8       loop
9           SESSION.SESSION_LIST (L);
10          PUT (BOUNDED.IMAGE (L));
11          NL;
12          delay DURATION (3);
        end loop;
13      PUT ("test8 all done");
14      NL;
    exception
        when others =>
15          PUT ("Exception raised in main task! oh no!");
16          NL;
17          PUT (DEBUGGER.CURRENT_EXCEPTION_NAME);
18          NL;
19          SESSION.SHUTDOWN;
    end;
```

Several numbering rules are illustrated in this example:

- *Use* clauses are not numbered; they do not have a run-time representation.
- Blocks are numbered separately (for example, test8.6s).
- Statements in exception handlers are numbered continuously with preceding statements.

Several sample pathnames using the above source are:

```
TEST8.1d        -- Declaration L
TEST8.6.2       -- Put statement inside anonymous block
TEST8.10s       -- Put statement inside loop
```

Example with Tasking Constructs

Another example of statement and declaration numbering follows:

```
        package SESSION is
1           type TERMINAL is (VT100, DASHER, ADM3);
2           type ID is new INTEGER;
3           procedure INITIALIZE;
        end;
        package body SESSION is
            use TTY_IO, BOUNDED;
1           package CTIME is ...;
2           task MANAGER is ...;
3           task body MANAGER is separate;
        end;
        task body SESSION.MANAGER is
            ...
        begin
1           accept INITIALIZE do
        1       INIT;
            end;
2           loop
3               select
4                   accept INITIALIZE do
        1               INIT;
                    end;
                or
5                   accept SESSION_LIST ( ... ) do
        1               BUILD_LIST (LIST);
                    end;
                or
6                   accept SET_USER ( ... ) do
        1               COPY (STATE (S).NAME, USER);
        2               STATE (S).TERM := TERM;
                    end;
                or
                    when FINI =>
7                   terminate;
                end select;
            end loop;
        end;
```

Additional numbering rules are made clear in this example:

- Package specs and bodies are numbered separately.
- Accept arms of select statements are numbered along with statements at the same level as the enclosing select statement.
- Bodies of accept statements are numbered independently.

Several example pathnames using the above source are:

```
SESSION'body.2d      -- the declaration of task MANAGER
SESSION'spec.2d      -- the declaration of type ID
SESSION.MANAGER.6    -- accept SET_USER...
SESSION.MANAGER.6.2  -- STATE assignment in accept body
```

Example with Generics

One final example of statement and declaration numbering with generic units follows:

```
        package GTEST is
            generic
            ...
    1       package G1 is ...;
        end;
        package body GTEST is
    1       procedure GPROC_INSTANCE is new GPROC(T => integer);
    2       package body G1 is separate;
        end;
        generic
            type T1 is private;
        package GTEST.G1 is
    1       type TG1 is new integer;
    2       procedure G1P1 (X : T1);
        end;
        package body GTEST.G1 is
    1       X : T1;
    2       Y : integer := 1;
    3       procedure G1P1 (X : T1) is separate;
        end;
        procedure GTEST.G1.G1P1 (X : T1) is
    1       procedure GP_INSTANCE is new GPROC (T => TG2);
        begin
    1       D.PUT ("G1P1 says hi");
    2       Y := Y + 1;
    3       GPROC_INSTANCE (2, 34);
            declare
    1           A : TG1;
    4       begin
    1           GP_INSTANCE (4, 2);
    2           A := A + 12;
            end;
        end;
```

Additional numbering rules are made clear in this example:

- Declarations within a package, and declarations and statements within subprograms, are numbered independently.
- Generic formal parameters are not numbered.
- Declare blocks are treated as subprograms with respect to numbering. The name of the declare block in the above source code is GTEST.G1.G1P1.4.

Some example pathnames using the above source code are:

```
GTEST'spec.ld       -- package G1...
GTEST.G1.G1P1       -- procedure inside generic G1
GTEST.G1.G1P1.4.ld  -- declaration of A
```

package Debug

Package !Commands.Debug contains the specific types, procedures, and functions provided by the Rational Environment Debugger for interactive use.

The following package, !Tools.Debug_Tools, provides a programmatic interface to the Debugger.

procedure Activate

```
procedure Activate (Breakpoint : Natural);
```

Description

Activates a previously removed (deactivated) breakpoint(s).

The previously defined and removed breakpoint is reactivated; that is, it is allowed to interrupt execution. Note that breakpoints set while debugging one job can be activated again even if the job is aborted and debugging is initiated on a new job.

Parameters

`Breakpoint : Natural;`

Specifies which breakpoint to reactivate. A parameter value of 0 indicates all inactive breakpoints are to be reactivated.

Errors

A breakpoint may not be able to be activated for various reasons. For example, the task to which it applies may not exist, or the Ada unit the breakpoint was set in may have been modified since the program began executing.

If all breakpoints are being activated using the value 0 for the Breakpoint parameter, the numbers of the breakpoints activated successfully and the numbers of the breakpoints not activated successfully are displayed by the Debugger.

If an Ada unit has been modified, breakpoints cannot be activated in it until the job being debugged is terminated and redebugged.

References

procedure Break

procedure Remove

procedure Address_To_Location

```
procedure Address_To_Location (Address : String := "");
```

Description

Displays the source location corresponding to the address of the specified machine instruction.

This procedure is the inverse operation of the Location_To_Address procedure.

Parameters

```
Address : String := "";
```

Specifies the address of a machine instruction.

The form of this parameter depends on the target being debugged. For the R1000 target, the address should be of the form *"#segment, #offset"*. The segment specifies (in hexadecimal) the segment name of the space to be accessed. The offset specifies the location of the instruction in the segment.

Example

The command `Address_To_Location ("19901, 10")` displays:

```
Name:   .PRODUCER_CONSUMER.QUEUE.1d
PC = #19901, #10
```

References

procedure Location_To_Address

procedure Break

```
procedure Break (Location         : Path_Name  := "<SELECTION>";
                 Stack_Frame      : Integer    := Ø;
                 Count            : Positive   := 1;
                 In_Task          : Task_Name  := "";
                 Default_Lifetime : Boolean    := True);
```

Description

Creates a breakpoint at the specified location in the specified task.

By default, a breakpoint will be set at the selected location for all tasks.

The Location parameter is the primary means for specifying the location of the breakpoint. The Stack_Frame parameter provides a convenient means for specifying a frame in which to set a breakpoint. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Break command is bound to a key. If the Stack_Frame parameter is nonzero and the Location parameter specifies a special name (such as "<SELECTION>"), the Location parameter is ignored and a breakpoint is set in the first location of the frame specified by Stack_Frame. If the Stack_Frame parameter is nonzero and the Location parameter specifies a relative pathname, the actual pathname used to specify the location of the breakpoint will be composed by prepending the string *"_n"* to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Location parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If the Location parameter is the null string (""), or if a special name that does not resolve to a location is used (for example, if the Location parameter is "<SELECTION>" and the cursor is not in the selection), the breakpoint is set in the first location in the current frame.

Execution of that task stops before the execution of the specified statement or declaration. If the Count parameter is more than 1, the task does not stop the first Count–1 times that the location is executed.

If the Freeze_Tasks flag is true when a breakpoint is encountered, all other tasks will attempt to stop. Note that in this mode other tasks may not actually stop because they may be in rendezvous with the stopped tasks, waiting for an entry call, and so on.

A breakpoint is a location in the source where execution of the source code is interrupted if certain conditions are met. The first condition is that the breakpoint be active. When breakpoints are created, they are active; they can be deactivated (removed) and later reactivated at any time.

The second condition is that the location where the breakpoint is set has been executed a specified number of times (specified by the Count parameter). A breakpoint can be created that requires the breakpoint to be executed (passed) a specified number of times before it interrupts the execution of the task.

Breakpoints can be set to apply to a specific task only or to any task. The Break procedure creates an active breakpoint at the specified location in the specified task. If no task name is specified, the breakpoint will be set in the control context task or, if the control context is not set, all tasks, both current and future. If applied to a specific task, another task executing in the location specified in the breakpoint will be unaffected. See notes under the In_Task parameter, below.

A number, which must be used to delete or remove that breakpoint, is assigned to the new breakpoint. If more than 16 active breakpoints apply to a task at the same time, there is a heavy execution penalty.

When the conditions of a breakpoint are met, a message is printed that includes the breakpoint number and location. Execution is stopped before the specified location. Execution of that task awaits a command to continue executing. If the Freeze_Tasks flag is true, the Debugger attempts to stop all tasks when the breakpoint is encountered.

Breakpoints can be either temporary or permanent. The difference determines what happens to the breakpoint when it is triggered. A permanent breakpoint is left installed and activated, whereas a temporary breakpoint is deactivated and, optionally, deleted. Thus, a temporary breakpoint happens only once without additional user action.

The Permanent_Breakpoints option controls whether, by default, breakpoints are created as temporary or permanent by the Break procedure. By default, it is true. The value of the Default_Lifetime parameter determines whether the default lifetime established by the Permanent_Breakpoints option should be used when creating the breakpoint. Specifically, if the Default_Lifetime parameter is true, the breakpoint created will be permanent if the Permanent_Breakpoints option is true and will be temporary if the Permanent_Breakpoints option is false. If the Default_Lifetime parameter is false, the breakpoint created will be temporary if the Permanent_Breakpoints option is true and will be permanent if the Permanent_Breakpoints option is false. See Option type for more information on this option.

The Delete_Temporary_Breaks option causes temporary breakpoints to be deleted after they are deactivated as a result of being triggered. The standard value for this option is false.

The command `Show (Breakpoints)` displays the currently existing breakpoints and lists their status.

When a breakpoint is triggered by a task, the task stops execution and a message is displayed in the Debugger window. An example of this message is:

```
Break  2:  .RECURSIVE_CALLS.3s  [Task : ROOT_TASK, #674D8].
Breakpoint 2 deactivated.
```

The message specifies the breakpoint number, the location where the task has stopped, and the task name. It also indicates that breakpoint 2 was temporary and has been deactivated.

Parameters

```
Location : Path_Name := "<SELECTION>";
```

Specifies the location at which the breakpoint is to be set. The interpretation of this parameter is discussed in more detail in the description above. By default, the breakpoint will be set at the selected location for all tasks.

The pathname cannot specify entry declarations or delay or terminate statements in select arms.

If the name specifies a specific instance of a generic, the break occurs only in that instance. If it specifies a generic definition, the break occurs in any instance.

```
Stack_Frame : Integer := 0;
```

Specifies the frame in which to set the breakpoint. The interpretation of this parameter is discussed in more detail in the description above. By default (when Stack_Frame = 0), this parameter will be ignored and the breakpoint will be set at the location specified by the Location parameter.

```
Count : Positive := 1;
```

Specifies the number of times the breakpoint must be executed before it interrupts the execution of the task. The default specifies that the breakpoint interrupts execution the first time the breakpoint is executed.

```
In_Task : Task_Name := "";
```

Specifies the task in which the breakpoint is to be set. By default, the breakpoint will be created in the control context task or, if the control context is not set, in all tasks, both current and future.

The reserved string "all" means that the breakpoint should apply to all tasks, current and future, in the program.

```
Default_Lifetime : Boolean := True;
```

Specifies whether the breakpoint is to be permanent or temporary. By default, the breakpoint will be permanent (unless the Permanent_Breakpoints option has been set to false).

The value of the Default_Lifetime parameter determines whether the default lifetime established by the Permanent_Breakpoints option should be used when creating the breakpoint. Specifically, if the Default_Lifetime parameter is true, the breakpoint created will be permanent if the Permanent_Breakpoints option is true and will be temporary if the Permanent_Breakpoints option is false. If the Default_Lifetime parameter is false, the breakpoint created will be temporary if the Permanent_Breakpoints option is true and will be permanent if the Permanent_Breakpoints option is false. See Option type for more information on this option.

Restrictions

A maximum of 30 breakpoints for specific tasks can be set. A maximum of 20 breakpoints for "all" tasks can be set. The two sets are independent of each other. However, 16 active breakpoints of either kind are accelerated; more than 16 active breakpoints cause program execution to be degraded.

References

procedure Activate

procedure Execute

type Option

procedure Remove

procedure Show

---

procedure Catch

```
procedure Catch (Name         : Exception_Name  := "<SELECTION>";
                 In_Task      : Task_Name       := "";
                 At_Location  : Path_Name       := "");
```

Description

Stops execution whenever the named or selected exception is raised in the specified task(s) at the specified location; reports the task name, the location in which the exception was raised, and the exception name.

By default, when the selected exception is raised in any task, the task will stop in the Debugger.

The Name parameter names the specific exception or group of exceptions to be caught by the Debugger. If the Name parameter is the null string (""), or if a special name such as "<SELECTION>" is used but the cursor is not in the selection, the procedure causes execution to stop for all exceptions. The reserved string "all" means all exceptions. The reserved string "implicit" means exceptions raised implicitly—that is, those raised in the course of the execution of a statement other than raise.

The In_Task parameter specifies the task in which the exception should be caught. The reserved string "all" means that the catch request should apply to all tasks. The null string ("") means that the catch request should apply to the task specified by the current control context or to all tasks if the control context is not set to a specific task.

The At_Location parameter restricts the location where the exception, if it is raised, will be caught by the Debugger. The null string indicates everywhere the exception is raised. If not null, the string specifies a subprogram or statement where the catch request applies.

The Debugger maintains a list of catch and propagate requests entered by calls to the Catch and Propagate procedures. When an exception in the user program is raised, the Debugger looks at this list to determine whether to stop the program and inform the user. Catch requests cause the program to stop; propagate requests cause it not to stop.

If the Freeze_Tasks flag is true when the execution of a task is stopped because an exception is caught, then the Debugger attempts to stop all other tasks. Note that in this mode other tasks may not actually stop because they may be in rendezvous with the stopped tasks, waiting for an entry call, and so on.

The action taken in a specific case is determined by the most specific request that applies to the exception. If that request is a catch request, the program stops; otherwise, it does not stop.

Informally, a catch or propagate request applies to the raising of an exception if the exception name, task name, and location in the exception match the request. More precisely, a request applies to an exception if all of the following are true:

- The name of the exception is the same as the name in the request, the request is for all exceptions, or the request is for implicit exceptions and the exception was raised implicitly.
- The task in which the exception is raised equals the task in the request or the request is for all tasks.
- The exception is raised in a statement or declaration that the request specified, the point of raise is in a subprogram that the request specified, or the request is for all locations.

If more than one catch or propagate request applies to a specific raising of an exception in a program, the more specific request determines the action the Debugger will take. Requests are considered more specific if they specify a smaller number of cases. Thus, the more parameters of the request that are specified, the more specific the request. More formally:

- A request that specifies a subprogram and statement is more specific than one that specifies only a subprogram.
- A request that specifies a subprogram is more specific than one that specifies all locations.
- A request that specifies a task is more specific than one that does not.
- A request that names an exception is more specific than one that specifies implicit or all exceptions. A request that specifies implicit exceptions is more specific than one that specifies all exceptions.

These rules are applied in combination in the order given. The location is a stronger specification than the task and the task is a stronger specification than the exception.

The command `Catch ("all","","")` is the least specific and requests that exceptions not covered by other catch or propagate requests result in the Debugger stopping the task that raises an exception not covered in another request.

When the Debugger is started, it behaves as if the command `Catch ("all","","")` had been issued, meaning that all exceptions are to be caught in all locations for all tasks. If the Save_Exceptions option (see Option type) is true, exception-handling information will be saved from debugging run to debugging run; by default, this option is false.

To remove a catch or propagate request, use the Forget procedure.

If a catch request has parameters that exactly match a propagate request, then the propagate request is first removed.

The command `Show (Exceptions)` displays all catch and propagate requests ordered, from most specific to least specific. For each request, the exception name, location, and task restrictions are listed.

When a task stops because of an exception being raised, a message of the following form is displayed in the Debugger window:

```
Exception Constraint_Error caught at
    .TEST_CASE.4s  [Task : #1349704].
```

This message indicates the exception raised, the point in the program in which it is raised, and the name of the task that raised it.

In some cases, the name of the exception may not be available. For example, the exception may have been raised in Environment code, in code-archived code, or in a unit that has been modified since debugging of the program began. The exception name is then displayed either as a large hexadecimal number or as a string of the form <Unit=1234, Ord=2> or as a string of the form <Space=3, Index=234987>. This information can be of use in submitting problem reports to Rational. Note that the command Information (Exceptions) may provide additional useful information in these cases.

Parameters

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

Specifies the exception to be added to the catch list. By default, the selected exception will be added.

If Name is the null string (""), or if a special name such as "<SELECTION>" is used but the cursor is not in the selection, the procedure stops execution for all exceptions (equivalent to the reserved name "all").

The reserved name "all" stands for all exceptions.

The reserved name "implicit" stands for all exceptions raised implicitly—that is, raised by some language construct other than a raise statement. These implicit exceptions include only predefined language exceptions such as Constraint_Error and Tasking_Error.

If the exception name is not fully qualified (and nonnull), it is interpreted relative to the current evaluation context.

```
In_Task : Task_Name := "";
```

Specifies the task that should be monitored for the exception specified by the Name parameter. If null, the task specified by the control context is used. If the control context is not set to a specific task, the Catch procedure applies to all tasks. The reserved name "all" also implies all tasks. By default, exceptions will be caught when raised by any task, unless the control context is set.

If In_Task is prefixed with a percent symbol (%) or begins with a digit, then it names a specific task by means of its task number or its name assigned with a call to the Debug.Set_Task_Name procedure or the Debug_Tools.Set_Task_Name procedure. Otherwise, if the task name is nonnull and not fully qualified, the name is interpreted relative to the current evaluation context.

```
At_Location : Path_Name := "";
```

Specifies a location restriction for the exception catch request. The raising of the specified exception causes the specified task to stop only if the point of raise of the exception is in the subprogram or at the statement specified by the At_Location parameter. By default, exceptions will be caught anywhere they are raised.

If the At_Location parameter specifies an Ada unit, the exception is caught only inside that unit; it is not caught inside any Ada units nested inside it, including nested blocks and accept statements.

If At_Location specifies a particular statement or declaration, the exception is caught only when it is raised in the statement or declaration.

If At_Location is null, the catch applies anywhere.

Note: The evaluation context is used in interpreting At_Location only if it is not null. If At_Location is null, the catch request applies throughout the program, independent of the evaluation context.

If not fully qualified and not null, the Path_Name parameter is interpreted relative to the current evaluation context.

Restrictions

A maximum of 40 requests to catch or propagate exceptions can be set.

Example 1

A more restrictive specification always overrides a less restrictive one. A specification of a single task, exception, or location takes precedence over a specification of all tasks, all exceptions, or all locations, respectively. For example, if the commands:

```
Propagate (Name => "constraint_error", In_Task => "all",
       At_Location => "");
Propagate (Name => "all", In_Task => "all",
           At_Location => "!USERS.PHIL.TEST");
Catch (Name => "all", In_Task => "1349704",
       At_Location => "");
```

are issued, any exception raised in !Users.Phil.Test (the most specific request) does not cause the Debugger to stop program execution. The third request is more specific than the first, so the raising of the Constraint_Error exception causes the Debugger to stop program execution only if it is raised in task 1349704; any other exception raised in task 1349704 results in that task being stopped as well.

Example 2

The name of an individual exception is considered more restrictive than the name "implicit". Thus, the commands:

```
Catch (Name => "implicit", In_Task => "",
       At_Location => "");
Propagate (Name => "numeric_error", In_Task => "",
           At_Location => "");
```

cause all implicitly raised exceptions except Numeric_Error to be caught.

Note: The above Propagate procedure applies to all Numeric_Errors, whether implicitly or explicitly raised.

Example 3

A location specifying a statement or declaration within an Ada unit is more specific than a location specifying the entire Ada unit. Thus, the commands:

```
Catch (Name => "Status_Error", In_Task => "",
       At_Location => "Sandy.1");
Propagate (Name => "Status_Error", In_Task => "",
           At_Location => "Sandy");
```

result in the Status_Error exception, causing program execution to stop only if the exception is raised at statement 1 of subprogram Sandy.

Example 4

The commands:

```
Catch (Name => "all", In_Task => "",
       At_Location => "Sandy.Test");
Propagate (Name => "all", In_Task => "1349704",
           At_Location => "");
```

result in program execution stopping when any exception is raised by any task (including 1349704) inside Sandy.Test, because the catch request is more specific than the propagate request (the location is more specific than the task name).

Example 5

The commands:

```
Prop (Name => "all", In_Task => "1349704",
          At_Location => "");
Catch (Name => "Constraint_Error", In_Task => "",
       At_Location => "");
```

result in the Constraint_Error exception not stopping program execution in task 1349704 (but in all other tasks), because the propagate request is more specific than the catch request (a task specification is more specific than an exception specification).

References

procedure Forget

type Option

procedure Propagate

procedure Show (Exceptions)

---

procedure Clear_Stepping

```
procedure Clear_Stepping (For_Task : Task_Name := "");
```

Description

Removes all pending stepping operations that have been applied to the specified task or tasks.

By default, stepping operations are cleared for all tasks.

Any stepping (Run) conditions that have not been met are deleted. Typically, this command is used to remove unneeded step points. Removing step points allows the program to run faster because step points are no longer being checked for.

If Task_Name specifies all tasks, then stepping operations for all tasks are deleted.

Parameters

`For_Task : Task_Name := "";`

Specifies the task for which stepping should be canceled. The default is the task specified by the current control context. If the control context is not set to a specific task, the default is all tasks. The string "all" also represents all tasks.

Example

The command:

```
Clear_Stepping ("620E");
```

cancels any stepping operations that are in progress for task 620E. Note that the leading percent symbol (%) is optional.

procedure Comment

```
procedure Comment (Information : String := "");
```

Description

Displays the comment specified by the String parameter in the Debugger window.

The procedure can be used to place comments in the Debugger window to allow referring back to certain areas easily or to annotate the debugging session.

Parameters

```
Information :  String := "";
```

Specifies the string to be displayed in the Debugger window.

procedure Context

```
procedure Context (Set          : Context_Type  := Debug.Control;
                   To_Be        : Path_Name     := "<SELECTION>";
                   Stack_Frame  : Integer       := 0);
```

Description

Sets the specified context to be the specified pathname.

By default, the control context will be set to the selected item.

The Debugger evaluates unqualified pathnames, task names, and exception names in one of two contexts: the control context or the evaluation context. If the pathname, task name, or exception name is fully qualified, the context is not used.

If selection is used to specify the context, the stack is searched from the frame indicated by the value of the Stack_Start numeric option—by default, the top frame—to the bottom frame to find the first occurrence of the desired context. Note that if the Stack_Frame parameter is nonzero, this search will begin in the frame indicated by that parameter instead. The search is limited to the number of frames of the stack indicated by the Stack_Count numeric option—by default, 10 frames. If selection is not used, this searching is not performed.

The control context is used in the following commands:

- Break: To specify the default value for the task to which the break will be restricted. The control context also specifies which task's stack will be used if the pathname in the procedure refers to a stack.
- Catch: To specify the task for which exception controls should be set.
- Clear_Stepping: To specify the default task for which stepping operations should be canceled.
- Display: To specify which task's stack will be used if the pathname in the Display procedure refers to a stack.
- Execute: To specify the default task to be started.
- Forget: To specify the exception and the task for which exception operations will be cleared.
- History_Display: To specify the default task for which history information is to be displayed.
- Hold: To specify the default task to be held.
- Propagate: To specify the task for which exception controls should be set.
- Put: To specify which task's stack will be used if the pathname in the Put procedure refers to a stack.
- Release: To specify the default task to be released.

- Run: To specify the default task to be stepped.
- Stack: To specify the default task whose stack will be displayed.
- Stop: To specify the default task to be stopped.
- Take_History: To specify the default task for which history information is to be gathered.
- Trace: To specify the default task for which a tracing operation will be enabled or disabled.

The control context can be set to the null string (""). In this case, the above commands default to all tasks where a task to apply to an operation is needed; they default to the last task stopped where a specific task's stack is needed. Specifically, the last task stopped is used for interpreting pathnames in any command and as the Task_Name parameter in the Stack and Run commands. All other uses reference "all" tasks.

The evaluation context is used in the following commands:

- Break: To specify a context in which to interpret unqualified object names.
- Catch: To specify a context for unqualified location and exception names.
- Context: To specify a context for unqualified names for the control or evaluation context.
- Display: To specify a context in which to interpret unqualified object names.
- Forget: To specify a context for unqualified location and exception names.
- Modify: To specify a context in which to interpret unqualified object names.
- Propagate: To specify a context for unqualified location and exception names.
- Put: To specify a context in which to interpret unqualified object names.
- Take_History: To specify a context for unqualified location names.
- Trace: To specify a context for unqualified location names.

As noted above, if the pathname given to any of these commands is fully qualified, the evaluation context is not used. If the name is not qualified—that is, if it does not begin with a period (.), underscore (_), exclamation mark (!), dollar sign ($), double dollar sign ($$), caret (^), or percent symbol (%)—the evaluation context is prepended to the pathname with appropriate connecting punctuation.

The current control and evaluation contexts are displayed by issuing the command `Show (Contexts)`.

Parameters

```
Set : Context_Type := Debug.Control;
```

Specifies which context to set (either Control or Evaluation).

```
To_Be : Path_Name := "<SELECTION>";
```

Specifies the value for the context. By default, the context is set to the selected item.

The context must be valid at the time it is specified and at the time it is used. For example, an evaluation context of _4.X must be valid when specified (frame 4 must exist and have an object X in it) and when it is used (command `Put ("Y")` can be issued only when _4.X.Y is legal).

The control context can be specified as an Ada pathname (which must reference a task), a task number, or a task nickname. The nickname is established by a task by calling the Debug.Set_Task_Name procedure or the Debug_Tools.Set_Task_Name procedure. If the control context is not a task number, nickname, or fully qualified name, it is interpreted in the current control and evaluation context as any other pathname. This interpretation is done before the new value for the control context is established. Thus, the name given to the Context procedure uses the current context in the calculation of the new one.

The control context can also be set to the null string ("") or the reserved name "all", both of which mean all tasks.

```
Stack_Frame : Integer := 0;
```

Specifies the stack frame containing the context or that is the context. Note that the stack frame can also be specified in the To_Be parameter; this parameter typically is used when its value is provided using argument prefix keys when the Context command is bound to a key. If the Stack_Frame parameter is nonzero and the To_Be parameter specifies a relative pathname, the actual pathname used to specify the context is composed by appending the string "_*n*" to the value of the To_Be parameter, where *n* is the value of the Stack_Frame parameter. If the To_Be parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

Restrictions

The pathname must be legal at the time the command is issued and at the time the context is used by another command.

Example

Consider the following commands:

```
Context (Control, "%Session_Manager");
Stop;
Context (Evaluation, "_3.Session_Map");
Show (Contexts);
Context (Control, "%340E");
```

In the first command, Session_Manager is presumed to be a task nickname. The evaluation context is then set to a package Session_Map, for example, declared in the subprogram running in stack frame 3 of that task. Note that the task is first stopped in the second command. The Stop procedure stops the task specified by the control context given in the first command. It is necessary to stop the task in order to process a pathname such as _3.Session_Map; otherwise, the stack could change at any time.

Finally, the contexts are displayed, and the control context is changed to a task with number 340E. Note that this change affects the evaluation context, which is unlikely to be valid at this point because frame 3 of task 340E probably does not contain an object named Session_Map.

type Context_Type

```
type Context_Type is (Control, Evaluation);
```

Description

Defines which type of context is set by the Context procedure.

Context type is used in the Context procedure only to select which context is to be set. See the discussion under the Context procedure for the effect of the control and evaluation context.

Enumerations

`Control`

Specifies a task for use as the default task in commands for which a task should be specified.

`Evaluation`

Specifies a prefix for unqualified pathnames in commands that specify a pathname.

References

procedure Context

procedure Convert

```
procedure Convert (Number  : String  := "";
                   To_Base : Natural := 0);
```

Description

Converts the string specified in the Number parameter to the specified base representation; 64-bit arithmetic is used.

By default (To_Base not specified), a decimal representation is converted to hexadecimal, and a number representation with a specific base is converted to decimal.

Parameters

```
Number : String := "";
```

Specifies the string representation of the number to be converted. The number is assumed to be in decimal if it does not include a base specification. A leading number symbol (#) indicates a hexadecimal number. The number can also be an Ada style-based number—for example, 8#177400#. Legal bases are 2 through 16.

```
To_Base : Natural := 0;
```

Specifies the base to which the number is to be converted. Legal values are 2 through 16. If unspecified (0), the base is decimal if the Number parameter is not decimal, and it is hexadecimal if the number is decimal.

procedure Current_Debugger

```
procedure Current_Debugger (Target : String := "");
```

Description

Causes the named debugger to become the current default debugger for the user's session.

This command is used when there are debuggers for various targets active in the session along with the R1000 Native Debugger. See the documentation for Rational's cross-development facilities for more information on using debuggers for these targets.

Parameters

`Target : String := "";`

Specifies the name of the image of the debugger desired. If the value is the null string (""), the default, the procedure brings the current Debugger window onto the screen. If it is the null string and in a Debugger window, that debugger becomes the current default debugger.

procedure Debug_Off

```
procedure Debug_Off (Kill_Job : Boolean := False);
```

Description

Terminates debugging of the current job.

The job being debugged is either released to continue normal execution or is aborted based on the value specified in the Kill_Job parameter.

This Debug_Off procedure differs from the Debug_Off procedure in package Debug_Tools. The procedure in package Debug_Tools terminates debugging only if the task calling Debug_Tools.Debug_Off is part of the job being debugged, and then it disables debugging only for that task.

Parameters

`Kill_Job : Boolean := False;`

Specifies, if true, that the job being debugged should be aborted. If false, the default, the job continues normal execution.

References

procedure Debug_Tools.Debug_Off

renamed procedure Disable

```
procedure Disable (Variable : Option;
                   On       : Boolean := False) renames Enable;
```

Description

Enables or disables the option flag controlling the behavior of the Debugger specified by the Variable parameter.

By default, the specified option is disabled.

See the Option type for more information on the option flags and their meanings.

Parameters

`Variable : Option;`

Specifies the option to be enabled or disabled.

`On : Boolean := False;`

Specifies whether to enable (turn on) or disable (turn off) the option. If unspecified, the default is to turn off the option.

References

procedure Enable

type Option

procedure Display

```
procedure Display (Location    : Path_Name := "<SELECTION>";
                   Stack_Frame : Integer := 0;
                   Count       : Natural := 0);
```

Description

Displays an area of source in the Debugger window with statement numbers included, based on the current selection or the pathname provided.

By default, the Display procedure displays the source for the selected item.

The Location parameter references some location in the program or data. If a data object or variable is referenced, the type declaration of that object is displayed. If a type is referenced, the type declaration is displayed. If an Ada program unit is referenced, the source (statements and declarations) of that unit is displayed.

The Location parameter is the primary means for specifying the source to display. The Stack_Frame parameter provides a convenient means for specifying a frame to display. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Display command is bound to a key. If the Stack_Frame parameter is nonzero and the Location parameter specifies a special name (such as "<SELECTION>"), the Location parameter is ignored and the source for the frame specified by Stack_Frame is displayed. If the Stack_Frame parameter is nonzero and the Location parameter specifies a relative pathname, the actual pathname used to specify the object to display is composed by appending the string *"_n"* to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Location parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If the Location parameter is the null string (""), or if a special name that does not resolve to an object is used (for example, if the Location parameter is "<SELECTION>" and the cursor is not in the selection), the current location in the current frame is displayed.

When program units are displayed, the code is formatted with declaration and statement numbers. These statement and declaration numbers are used in various other operations such as the Break and Trace procedures.

When a type is displayed, the various characteristics of the type are displayed even if the type is private. The one exception to this is when a private type is defined in some Environment-supplied code, in which case the Debugger will not display anything about the type.

If the pathname refers to an area of code that is in execution, the asterisk (*) or the number symbol (#) will appear next to a statement or declaration. The asterisk identifies the statement that the current task is currently executing (or about to execute). The number symbol identifies statements or declarations that are in execution in subprogram frames other than the topmost frame.

Note: The current task is the task specified by the control context or is the last task to stop if the control context is not set.

If a declaration is named, only declarations are displayed.

Parameters

```
Location : Path_Name := "<SELECTION>";
```

Specifies a location in the source to display. The interpretation of this parameter is discussed in more detail in the description above. By default, the source for the selected item is displayed.

The form of the name determines some aspects of the display.

If a specific statement is named, the display starts with that statement and continues for the number of statements specified by the Count parameter or until the end of the subprogram.

If a specific declaration is named, the display starts with that declaration and continues for the number of declarations specified by the Count parameter or until the end of the declaration list.

If a package, task, subprogram, or frame is named, the display includes the first declarations and statements in that unit. The Count parameter limits the number of declarations and the number of statements.

If a frame is displayed, the calling location is indicated with the * character for the top frame or the # character for lower frames.

The Declaration_Display (Option type) controls whether declarations are displayed as part of the display. Its standard value is true, which causes declarations to be displayed.

```
Stack_Frame : Integer := 0;
```

Specifies the frame for which to display the source. The interpretation of this parameter is discussed in more detail in the description above. By default, this parameter is ignored, and the item specified by the Location parameter is displayed.

```
Count : Natural := 0;
```

Specifies the number of statements and/or declarations to be displayed. Up to Count declarations and Count statements are displayed.

If the parameter is 0 (the default), the value used is determined by the value of Display_Count (Numeric type). The standard value of Display_Count is 10.

Restrictions

If a task is executing code that is part of the Environment or was compiled without debug tables, the source for that part of the code is not available for display.

Note: Debug tables, produced by the Environment's compilation system, contain information needed by the Debugger. It is possible to turn them off, but they are produced by default.

References

procedure Context

type Numeric, enumeration Display_Count

type Option, enumeration Declaration_Display

procedure Source

procedure Debug_Tools.Register

---

procedure Enable

```
procedure Enable (Variable : Option;
                  On       : Boolean := True);
```

Description

Enables or disables the option flag controlling the behavior of the Debugger specified by the Variable parameter.

By default, the specified option is enabled.

See the Option type for more information on the option flags and their meanings.

Parameters

`Variable : Option;`
Specifies the option to be enabled or disabled.

`On : Boolean := True;`
Specifies whether to enable (turn on) or disable (turn off) the option. If unspecified, the default is to turn on the option.

References

procedure Disable

type Option

subtype Exception_Name

```
subtype Exception_Name is String;
```

Description

Defines a string name for exceptions.

Several operations require a specific exception to be named. This type defines the way in which the exception is named. The name is an Ada identifier evaluated in the current control/evaluation context, a pathname (see subtype Path_Name for more information on pathnames), or a *special name* (see below). Two strings are predefined: the null string ("") means all exceptions, and the string "implicit" means all exceptions that are raised implicitly. An implicitly raised exception is an exception raised during the course of executing a statement or elaborating a declaration rather than an exception raised explicitly with a raise statement.

Exception names can be *special names* that indicate that the selection, image, cursor, and so on are to be used to specify the exception. These special names have the form "<SELECTION>" and are described in more detail in LM, Key Concepts. Note that if a special name does not resolve to an exception (for example, if the special name is "<SELECTION>" and the cursor is not in the selection), the exception name used by the Debugger will be the empty string ("").

A number of predefined exceptions are defined by the Ada language in package Lrm.Standard or are defined by the R1000 architecture. Exceptions defined by the R1000 architecture are for Rational internal use only and should not occur in any user programs. Predefined exception names are used without any qualification; they are entered exactly as they are listed below.

Exceptions predefined by package Lrm.Standard as they must be entered are:

```
Constraint_Error        Storage_Error
Numeric_Error           Tasking_Error
Program_Error
```

When the Debugger displays a predefined exception name, it may include additional information in parentheses following the exception name. This information more precisely indicates the nature of the exception. The information in parentheses should not be entered when you name an exception.

The following list shows the exception names as displayed by the Debugger and the states that caused the exception:

- Constraint_Error (Array Index): Attempt is made to access an array element using a subscript that is out of the range of the array index type.
- Constraint_Error (Case Range): Case expression is out of bounds of the case statement (this exception can happen only in erroneous Ada programs).

- Constraint_Error (Discriminant): Attempt is made to access a field of variant record that is not present in the specific record being accessed.
- Constraint_Error (Entry Family): Use of an entry family index that is out of range.
- Constraint_Error (Exponent): Integer is raised to a negative exponent.
- Constraint_Error (Length Error): Result of a concatenation that has an upper bound greater than that of the index subtype (see the *Reference Manual for the Ada Programming Language*, Section 4.5.3).
- Constraint_Error (Null Access): Attempt is made to dereference a pointer that is null.
- Constraint_Error (Type Range): Conversion of a value to a type that does not include the value in its range.
- Numeric_Error (Overflow): Evaluating an expression that is not representable; often uninitialized *out* parameters.
- Numeric_Error (Zero Divide): Attempt to divide by zero.
- Program_Error (Elaboration Order): Access to a program unit not yet elaborated.
- Program_Error (Function Exit): "Falling off" the end of a function without execution of a return statement.
- Program_Error (Prompt Executed): Execution of an Editor prompt (typically a `[statement]` xeprompt).
- Program_Error (Select): Execution of a select statement with no open alternatives.
- Storage_Error (Allocation): Reserved for future use.
- Storage_Error (Control): Too deeply nesting subprogram calls in a task (probably an infinite, recursive loop).
- Storage_Error (Data): Allocation of too many objects. (When an access type is declared, a maximum amount of space is allocated for all objects designated by the access type. Once this space is filled, Storage_Error (data) results on all further allocations for this type. Other access types may still have available space. A pragma specifies the amount of space to be reserved for an access type.)
- Storage_Error (Import): Too many objects referenced by a package, task, or subprogram subunit.
- Storage_Error (Name): Too many tasks or packages declared.
- Storage_Error (Oversize Object): An object greater than the maximum size allowed declared.
- Storage_Error (Program): Too much code required by a job.
- Storage_Error (Queue): Too many outstanding entry calls queued for a task.
- Storage_Error (Resource): Reserved for future use.
- Storage_Error (Type): Definition of too many types.
- Tasking_Error (Abnormal Task): Raised in a task when it is in rendezvous with

another task and that other task is aborted.

- Tasking_Error (Activation): Raised when an exception propagates out of a child task during its activation.
- Tasking_Error (Completed Task): Task that has terminated is accessed; for example, an attempt to make an entry call to a completed task.

References

procedure Catch

procedure Forget

subtype Path_Name

procedure Propagate

---

procedure Exception_To_Name

```
procedure Exception_To_Name (Implementation_Image : String := "");
```

Description

Displays the source name of the exception that corresponds to the specified implementation-dependent representation.

For the R1000 target, the Debugger occasionally may not implicitly know the name of an exception when it is raised; when this happens, the Debugger displays a space and an index pair of numbers. This procedure allows the Debugger to try to interpret those numbers.

Parameters

```
Implementation_Image : String := "";
```

Specifies an implementation-dependent representation of an exception.

The parameter is interpreted based on the target architecture on which the job is being debugged. For the R1000 target, the image should be of the form *"#segment, #offset"*. The segment specifies (in hexadecimal) the segment name of the space to be accessed. The offset specifies the starting location in the segment. If the number is preceded by a # character, or if it contains a character from *a* through *f*, it will be interpreted as a hexadecimal number instead of a decimal number. Note that a predefined exception can be specified by just one number (the exception's number) or by two numbers (0 and the exception's number). In general, the input can be of any form that the Environment prints out.

References

procedure Catch

procedure Execute

```
procedure Execute (Name : Task_Name := "");
```

Description

Commences (or resumes) execution of the named task or tasks.

This command is functionally equivalent to the Xecute procedure.

The named task starts executing from its current location—that is, from where it was stopped because of a breakpoint or a stop or hold request, an exception being trapped, or the end of a stepping request. If the task is executing, the procedure has no effect.

If a specific task is named and that task is being held (by means of the Hold procedure), then the hold condition is removed.

If the Task_Name parameter is "all", any task that is stopped for any reason in the Debugger is allowed to continue unless the Debugger has a hold on the task. Tasks subject to hold conditions must be started individually by name, or the hold condition must be released with the Release procedure.

If the Freeze_Tasks flag is true and all tasks are stopped implicitly as a result of an individual task being stopped, the Execute procedure will commence execution of these implicitly stopped tasks.

Parameters

`Name : Task_Name := "";`

Specifies the task to commence execution. The default is the task specified by the control context or all nonheld tasks if the control context is not explicitly set.

The reserved word "all" can be used to specify that all nonheld tasks are to be executed.

See the Hold procedure for more information on the held state.

Errors

A `No tasks are stopped` message occurs when no tasks are stopped in the Debugger or when the only ones stopped are subject to hold conditions.

procedure Execute
package !Commands.Debug

References

procedure Break

procedure Hold

procedure Release

procedure Stop

procedure Xecute

---

procedure Flag

```
procedure Flag (Variable : String := "";
                To_Value : String := "TRUE");
```

Description

Sets a flag controlling the behavior of the Debugger to a specified string value; the Variable parameter specifies the name of the flag and the To_Value parameter specifies its new value.

Flags in the Debugger control certain functions and provide special facilities. Flag names can be uppercase or lowercase; case is ignored.

Some of these flags are for Rational internal use only, not for general use; they are also unsupported.

The flags that can be set include string names for the enumeration values defined by the Option and Numeric types, as well as the following string names:

- Cache_Stack_Frames: Maintains a cache of current stack information for the task. Some Debugger functions are impaired when this flag is disabled. The standard value is true. Unsupported.
- Flag_Errors: Causes error messages to include ***.
- Hex_Values: Causes certain values to be displayed in hexadecimal from the Put procedure.
- Interpret_Import_Words: Causes the Memory_Display procedure to interpret words in the import space; default is false. Unsupported.
- Interpreter_Dump: Dumps the interpreter stack after each instruction executed by the interpreter; set to true or false. Unsupported.
- Interpreter_Trace: Prints information about execution of interpreted code used to do object fetch; set to true or false. Unsupported.
- No_Pointers: Causes pointer values to be listed as ppppp. The standard value is false.
- No_Task_Numbers: Causes task numbers to be listed as xxxxx rather than as actual numbers. The standard value is false.

Typically, the Boolean option flags (those defined by the Option type) would be enabled and disabled with the Enable and Disable procedures instead of the Flag procedure. Also, numeric flags (those defined by the Numeric type) typically would be set using the Set_Value procedure.

Parameters

```
Variable : String := "";
```

Specifies the name of the flag to be set.

```
To_Value : String := "TRUE";
```

Specifies the value to which the flag should be set.

Errors

No error checking is done on flag names or values. Illegal values can cause unexpected behavior of operations that depend on the flag values.

---

procedure Forget

```
procedure Forget (Name        : Exception_Name  := "<SELECTION>";
                  In_Task     : Task_Name       := "";
                  At_Location : Path_Name       := "");
```

Description

Removes catch and propagate requests that match the Name, In_Task, and At_Location parameters.

By default, catch and propagate requests are removed for the selected exception in all tasks and locations.

All catch and propagate requests are checked, and any that match the parameters to the Forget procedure are deleted. All three parameters must match according to the following rules:

- The exception name matches if the names are identical or if the Name parameter to Forget is the reserved name "all". If the Name parameter is the null string (""), the following error message is issued: `Name must be non-null; use "all" for all exceptions.`
- The task name matches if the name in the request and the In_Task parameter specify the same task or if In_Task refers to all tasks. This is the case if In_Task is the reserved string "all" or is null and the control context refers to all tasks.
- The location matches if the pathnames refer to the same location or if At_Location is null.

A Forget request with a name that is the reserved name "all", together with null task and location, clears all requests except one: the last command, either `Catch ("all")` or `Propagate ("all")`, with null task and location. This last entry still specifies how the Debugger is to treat exceptions raised in the program.

Parameters

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

Specifies an exception name to be used in matching catch and propagate requests to be removed. The reserved name "all" matches all exceptions. Any requests that also match the Task_Name and Path_Name parameters are deleted. By default, requests are removed for the selected exception.

```
In_Task : Task_Name := "";
```

Specifies the task that should match for a request to be removed. If it is the null string (“”), the task specified by the control context is used. If the control context is not set to a specific task, the Forget procedure matches all tasks. The reserved name “all” also matches all tasks. By default, requests are removed for all tasks unless the control context has been set.

If not fully qualified and not null, the In_Task parameter is interpreted relative to the current evaluation context. Note that the task number is used to identify the task, not the pathname. Thus, the same name can refer to different tasks at different times if, say, the name includes a variable of the access type that references a task and that variable is changed.

```
At_Location : Path_Name := "";
```

Specifies a location restriction for deleting catch and propagate requests. If the At_Location parameter is null (“”), the Forget procedure matches catch and propagate requests independent of location restrictions. By default, catch and propagate requests at any location will be removed.

If At_Location specifies an Ada unit, catch and propagate requests for the whole unit only are forgotten, but any requests for statements or declarations within the unit remain.

At_Location is interpreted relative to the current evaluation context if not null and not fully qualified.

References

procedure Catch

procedure Propagate

procedure Show (Exceptions)

---

subtype Hex_Number

```
subtype Hex_Number is String;
```

Description

Defines a string representation of a number in base 16.

Note: The number is assumed to be in hexadecimal; no "16#" need precede it.

procedure History_Display

```
procedure History_Display (Start    : Integer   := 0;
                           Count    : Integer   := 0;
                           For_Task : Task_Name := "");
```

Description

Displays a range of history entries for the specified task.

The Debugger can keep a history of execution for each task. The history is the set of the most recently executed statements and declarations.

History information, like traces, can include messages about statements, calls, rendezvous, and exception raising. Trace messages are displayed each time an event occurs. Unlike traces, history messages are saved in a circular buffer in the Debugger. From the buffer, selected sets of messages can be displayed, such as messages from only a specific task or for some range of messages. See the Trace procedure for more information on tracing. See the Take_History procedure for more information on histories.

See the example below for a discussion of the output form.

Parameters

```
Start : Integer := 0;
```

Specifies the starting entry in the history to be displayed. If greater than 0, it specifies the starting location in the history from the newest entry. If less than 0, it specifies the starting location from the oldest entry in the set. If unspecified or 0, the starting point is based on the value of History_Start (Numeric type), which is 10 by default.

```
Count : Integer := 0;
```

Specifies the number of entries in the history to be displayed. If unspecified, the number of entries displayed is determined by the value of History_Count (Numeric type), which is 10 by default.

```
For_Task : Task_Name := "";
```

Specifies the task for which the history is to be displayed. The default is the task specified by the control context or, if the control context is not set to a specific task, all tasks.

Example 1

The following is an example of a history display:

```
History of statements executed by all tasks  : (oldest .. newest)
  Timestamp  Depth Location and Task
 34656136386    2 .BUFFER.LENGTH.1d  [QUEUE, #2FØD4]
 +     284      1 .PRODUCER_CONSUMER.CONSUMER.5s   [CONSUMER, #2F8D4]
 +     522      2 .BUFFER.LENGTH.2d  [QUEUE, #2FØD4]
 +    1136      2     ....1s
 +    1435      2     ....2s
 +    1705      1 .PRODUCER_CONSUMER.QUEUE.7s
 +    1992      1     ....8s
 34657416781    1 .PRODUCER_CONSUMER.PRODUCER.3s   [PRODUCER, #2F4D4]
 34657816737    1 .PRODUCER_CONSUMER.CONSUMER.3S   [CONSUMER, #2F8D4]
 34658504120    1 .PRODUCER_CONSUMER.QUEUE.9s   [QUEUE, #2FØD4]
```

The first line describes what part of the history will be displayed and in what order. In this case, the statements from all tasks will be displayed, from the oldest entry to the newest entry.

The second line provides column headings. The first column is a measure of time in units of 3.2 microseconds. The second column is the depth on the stack at which the task, named in column 3, is currently running. The last column specifies the statement or declaration that the named task is executing.

Note: It takes the Debugger between 280 and 300 units in time to record each entry in the history. Time is monotonic for all tasks.

The remaining lines provide the data requested. By default, the Debugger displays 10 entries. Note that the statement or declaration name is elided if it is the same as in the previous line.

Example 2

The same history as in the previous example could be filtered to display only the history of one task. The following display occurs when only the task named Queue is requested:

```
History of statements executed by QUEUE, #2FØD4  : (oldest .. newest)
  Timestamp  Depth Location
 34656135064    2 .PRODUCER_CONSUMER.QUEUE.(accept  GET).2s
 +     274      1 .PRODUCER_CONSUMER.QUEUE.4s
 +     714      1     ....7s
 +    1322      2 .BUFFER.LENGTH.1d
 +    1844      2     ....2d
 +    2458      2     ....1s
 +    2757      2     ....2s
 +    3027      1 .PRODUCER_CONSUMER.QUEUE.7s
 +    3314      1     ....8s
 34658504120    1     ....9s
```

Note that the task is identified in the first line, not in each line of data. Also note that the history is searched further back in time to find the default 10 entries to display.

References

type Numeric

procedure Take_History

procedure Trace

type Trace_Event

---

procedure Hold

```
procedure Hold (Name : Task_Name := "");
```

Description

Stops execution of the specified task (or tasks) and keeps it stopped until the task is explicitly released by the Release procedure or until an explicit request is given for execution of the task by an Execute or Run procedure.

The task is held following the currently executing statement in that task, as in the Stop procedure. The Hold procedure allows for rendezvous and complex statements to complete before the hold takes effect. A message is displayed for each task when it stops. Tasks that are executing delays or waiting for rendezvous do not stop until those operations are complete.

If the Freeze_Tasks flag is true, the Debugger attempts to stop all other tasks. Note that in this mode other tasks may not actually stop because they may be in rendezvous with the stopped tasks, waiting for an entry call, and so on.

If the task is already held, the procedure has no effect.

Holding a task means that the user must issue an explicit command to make the task eligible for execution. The command can be issued in three ways. One way is to execute a Release procedure for the task (or all tasks). The other ways are to execute an Execute or a Run procedure for the specific task. Note that the command `Execute ("all")` does not make the help task executable.

The Hold procedure is used to take tasks out of the normal set of tasks running in the program. For example, if one task is misbehaving, from the point of view of the user debugging the program, it can be held. Debugging can then continue without interference from that task.

By using the command `Hold ("all")` followed by the Execute procedure for individual tasks, the user can debug the interaction of only the few individual tasks while the others remain held.

Parameters

`Name : Task_Name := "";`

Specifies what task is to be held. The default is the task specified by the current control context. If the control context is not explicitly set, the default is all tasks.

The string "all" can be used to specify that all tasks should be held.

References

procedure Execute

procedure Release

procedure Run

procedure Stop

---

procedure Information

```
procedure Information (Info_Type : Information_Type := Debug.Exceptions;
                       For_Task  : Task_Name        := "");
```

Description

Lists information about the specified task.

The Info_Type parameter specifies the type of information to be listed. If Info_Type specifies Exceptions, information about active exceptions in the specified task is listed. An exception is active if the task is still executing code in the handler for that exception. Information listed includes the exception name and the Ada name of the location where the exception was first raised.

If Addresses (Option type) is enabled, the program counter value, actual exception name, and control offset of the exception variable are also listed.

Note: The control offset of the exception variable is Environment information that contains the actual exception information. This information is generally useful only in diagnosing system problems.

If the Info_Type parameter specifies Rendezvous, information is listed about rendezvous that are in progress for the specified task. A rendezvous is in progress if the task is currently executing an accept statement for an entry. The information consists of the name of the rendezvous partner, which executed the entry call that resulted in the rendezvous.

If Addresses is enabled, the control offset of the task linkage is also displayed.

If the Info_Type parameter specifies Space (Information_Type type), information about the space that the task is consuming is displayed. The information includes the current and maximum control space and data space.

If Addresses is enabled, the address space sizes are also displayed.

Parameters

```
Info_Type :  Information_Type := Debug.Exceptions;
```

Specifies the type of information to be listed.

```
For_Task : Task_Name := "";
```

Specifies the task for which information is to be listed. If none is specified, the default (“”), information about the task specified by the current control context is listed. If the control context does not specify an individual task, information about all tasks is listed.

References

type Information_Type

type Option

---

type Information_Type

```
type Information_Type is (Exceptions, Rendezvous, Space);
```

Description

Defines the information that can be displayed by the Information procedure.

Enumerations

Exceptions
Specifies information about exceptions that are active in the specified task.

Rendezvous
Specifies information about rendezvous that are in progress in the specified task.

Space
Specifies information about the amount of memory consumed by the specified task.

procedure Kill

```
procedure Kill (Job      : Boolean := True;
                Debugger : Boolean := False);
```

Description

Kills the job being debugged and/or the Debugger for the session.

This command can be used to explicitly kill the job currently being debugged. This is the most typical use of this command.

The command can also be used to kill the Debugger for the session if the Debugger becomes unresponsive or misbehaves so that it is not possible to continue doing useful work. Note that the Debugger should not need to be killed in normal use.

The next time a job is created with debugging, a new copy of the Debugger is created. If the Debugger is killed, the job being debugged should run to completion (unless it is to be killed also), but the results may be unpredictable. If the Debugger is misbehaving, it is generally safest to use the command `Debug_Off (False)` to run the job to completion and then use the Kill procedure to kill the Debugger.

Parameters

`Job : Boolean := True;`

Specifies whether to kill the job currently being debugged. If true, the job is killed. If false, this command should have no effect on the job being debugged. By default, the job being debugged is killed.

`Debugger : Boolean := False;`

Specifies whether to kill the Debugger for the session. If true, the Debugger is killed. If false, the Debugger is not killed. By default, the Debugger is not killed.

procedure Location_To_Address

```
procedure Location_To_Address (Location    : Path_Name := "<SELECTION>";
                               Stack_Frame : Integer   := 0);
```

Description

Displays the code segment address for the machine instruction associated with the specified location.

The procedure displays the program counter (which consists, for the R1000 target, of a segment and an offset) where the machine instruction associated with the specified location exists in memory.

This procedure is the inverse operation of the Address_To_Location procedure.

The Location parameter is the primary means for specifying the location for which to determine the address. The Stack_Frame parameter provides a convenient means for specifying a frame for which to get the address. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Location_To_Address command is bound to a key. If the Stack_Frame parameter is nonzero and the Location parameter specifies a special name (such as "<SELECTION>"), the Location parameter is ignored and the address is determined for the first location of the frame specified by Stack_Frame. If the Stack_Frame parameter is nonzero and the Location parameter specifies a relative pathname, the actual pathname used to specify the location to get the address of is composed by appending the string *"_n"* to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Location parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If the Location parameter is the null string (""), or if a special name that does not resolve to a location is used (for example, if the Location parameter is "<SELECTION>" and the cursor is not in the selection), the address is determined for the first location in the current frame as determined by the control and evaluation contexts).

Parameters

```
Location : Path_Name := "<SELECTION>";
```

Specifies the location whose address is desired. The interpretation of this parameter is discussed in more detail in the description above. By default, the address is determined for the selected location.

```
Stack_Frame :  Integer := 0;
```

Specifies the frame whose address is to be displayed. The interpretation of this parameter is discussed in more detail in the description above. By default, this parameter is ignored and the address displayed is at the location specified by the Location parameter.

Example

The command `Location_To_Address` with the default parameter displays:

```
Name:   .PRODUCER_CONSUMER.QUEUE.1d
PC = #19901, #10
```

References

procedure Address_To_Location

---

procedure Memory_Display

```
procedure Memory_Display (Address : String  := "";
                          Count   : Natural := 0;
                          Format  : String  := "DATA");
```

Description

Displays the contents of absolute memory.

The parameters are interpreted based on the target architecture on which the job is being debugged.

For the R1000 target, the Address parameter should be of the form *"#segment, #offset"*. The segment specifies (in hexadecimal) the segment name of the space to be accessed. The offset specifies the starting location in the segment. The unit of the offset depends on the Format parameter. The Format values and the corresponding unit of the offset are:

Control	Control word offset (unit is control stack words).
Typ	Type word offset (unit is type stack words).
Queue	Bit offset (the display always starts on a full word boundary; the starting offset used will be the nearest full word that contains the bit specified by the Offset parameter).
Data	Bit offset into the segment (the display always starts on a full word boundary; the starting offset used will be the nearest full word that contains the bit specified by the Offset parameter).
Import	Word offset (unit is import words).
Code	Instruction offset (unit is instructions).
System	Bit offset (the display always starts on a full word boundary; the starting offset used will be the nearest full word that contains the bit specified by the Offset parameter).

Control words are interpreted based on their tag. Code segment words are disassembled into their symbolic form. All others are displayed in hexadecimal. (The Interpret_Control_Words flag can be set to false to disable the interpretation of control words. Import words are interpreted if the Interpret_Import_Words flag is set.)

Parameters

```
Address : String := "";
```

Specifies the address at which to display memory. The format of this parameter is interpreted differently for each target. See the description above for the interpretation for the R1000 target.

```
Count : Natural := 0;
```

Specifies the number of items to display. The meaning of this parameter is interpreted differently for each target. See the description above for the interpretation for the R1000 target.

```
Format : String := "DATA";
```

Specifies the format of the memory to be displayed. The meaning of this parameter is interpreted differently for each target. See the description above for the interpretation for the R1000 target.

---

procedure Modify

```
procedure Modify (New_Value   : String    := "";
                  Variable    : Path_Name := "<SELECTION>";
                  Stack_Frame : Integer   := 0);
```

Description

Modifies or changes the value of the specified object.

By default, the Modify procedure modifies the value of the selected object in the most recent frame of the stack of the last task to stop in the Debugger. If the object is declared in a library unit package, the object is determined from the package's state, not from a stack frame.

The specified object must be a scalar. Structures such as records or arrays must be modified component by component.

The Variable parameter is the primary means for specifying the object to modify. The Stack_Frame parameter provides a convenient means for specifying either the context for interpretation of the Variable parameter or the actual object to be modified. If the Stack_Frame parameter is nonzero and the Variable parameter specifies a relative pathname, the actual pathname used to specify the object to modify is composed by appending the string "_*n*" to the value of the Variable parameter, where *n* is the value of the Stack_Frame parameter. If the Variable parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If selection is used to specify the object, the stack is searched from the frame indicated by the value of the Stack_Start numeric option—by default, the top frame—to the bottom frame to find the first occurrence of the desired object. Note that if the Stack_Frame parameter is nonzero, this search will begin in the frame indicated by that parameter instead. This search is limited to the number of frames of the stack indicated by the Stack_Count numeric option—by default, 10 frames. If selection is not used, this searching is not performed.

If the Variable parameter is the null string (""), the procedure uses the current evaluation context to determine the object to modify. Note that this case is not useful unless the evaluation context is set to some variable; if it is not, the pathname will resolve, by default, to a frame (either the control context or the top frame of the last task stopped), which is not a variable.

Parameters

```
New_Value : String := "";
```

Specifies the new value of the object.

When modifying objects of numeric types, enter a string containing the simple numeric representation of the value. Expressions are not allowed.

When modifying objects of an enumeration type, enter a string containing the unqualified name of the enumeration constant. The object name is used to determine the type; therefore, qualification of the enumeration literal is unnecessary. Consistent with Ada, characters are treated as enumeration constants and entered surrounded by single quotes. (The New_Value parameter is a string, so the quoted character is actually inside the double-quoted string.) Recall that character literals such as Ascii.Nul are also entered unqualified (in this case, as Nul).

```
Variable : Path_Name := "<SELECTION>";
```

Specifies the object to be modified. See the description above and the reference entry for the Path_Name subtype for more information on how the names supplied to this parameter are interpreted. By default, the selected object is modified.

A number of restrictions define what can be modified, as listed under the New_Value parameter above.

```
Stack_Frame : Integer := 0;
```

Specifies the stack frame containing the variable to modify. Note that the stack frame can also be specified in the Variable parameter. If the Stack_Frame parameter is nonzero and the Variable parameter specifies a relative pathname, the actual pathname used to specify the object to display is composed by appending the string "_*n*" to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Variable parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

See the description above for more information on how this parameter is interpreted.

Restrictions

The pathname must specify an object that is a variable.

The specified object must be of a simple data type: discrete, float, or character.

Access types can be modified only to the null value.

The following objects cannot be modified:

- Variables of task types
- *In* parameters, which are actually constants
- Discriminants of variant records
- For-loop iteration variables
- Constants

Errors

The pathname cannot specify an object that is a constant.

The pathname cannot specify an object of a structured type. To modify structures, modify each of their scalar components one at a time.

References

procedure Put

type Numeric

```
type Numeric is (Display_Count, Display_Level, Element_Count,
                 First_Element, History_Count, History_Entries,
                 History_Start, Memory_Count, Pointer_Level,
                 Stack_Count, Stack_Start);
```

Description

Defines the numeric value flags that control the behavior of the Debugger that can be changed by the Set_Value procedure.

Note that Boolean option flags also control the behavior of the Debugger. See the Option type for more information on these flags.

In the following descriptions, the ***standard value*** of each numeric flag is listed. The standard value is the initial value of the numeric flag when the Debugger is started. The standard values for these numeric flags are read from session switches when the Debugger is started. The switches have names of the form Debug_*xxx*, where *xxx* is the numeric flag name. The standard values indicated below are the default values of these switches. See SJM, Session Switches, for more information on session switches.

Enumerations

`Display_Count`

Specifies the default value of the Count parameter in the Display procedure. The standard value is 10.

`Display_Level`

Specifies the number of levels to expand complex data structures in the Put procedure (the remaining levels are elided). The standard value is 3.

`Element_Count`

Specifies the maximum number of elements in any array that is displayed with the Put procedure. The standard value is 25.

`First_Element`

Specifies the first element of an array that is displayed with the Put procedure. The standard value is 0.

History_Count

Specifies the default value for the Count parameter to the History_Display procedure. The standard value is 10.

History_Entries

Specifies the maximum number of history entries that the Debugger will keep. The standard value is 1,000. This is not currently supported, so it has no effect.

History_Start

Specifies the oldest history entry to be displayed by the History_Display procedure. This enumeration is the default value for the Start parameter. The standard value is 10.

Memory_Count

Specifies the default value for the Count parameter in the Memory_Display procedure. The standard value is 3.

Pointer_Level

Specifies the level of pointer values to be expanded in the display produced by the Put procedure. The standard value is 3.

Stack_Count

Specifies the default value of the Count parameter in the Stack procedure. The standard value is 10.

Stack_Start

Specifies the default starting frame number in the Stack procedure. The standard value is 1.

References

type Option

procedure Set_Value

---

type Option

```
type Option is (Addresses, Break_At_Creation, Declaration_Display,
                Delete_Temporary_Breaks, Display_Creation, Echo_Commands,
                Freeze_Tasks, Include_Packages, Interpret_Control_Words,
                Kill_Old_Jobs, Machine_Level, No_History_Timestamps,
                Optimize_Generic_History, Permanent_Breakpoints,
                Put_Locals, Qualify_Stack_Names, Require_Debug_Off,
                Save_Exceptions, Show_Location, Timestamps);
```

Description

Defines the option flags controlling the behavior of the Debugger that can be enabled or disabled by the Enable or the Disable procedures.

Note that numeric flags also control the behavior of the Debugger. See the Numeric type for more information on these flags.

In the following descriptions, the *standard value* of each option is listed. The standard value is the initial value of the option when the Debugger is started. The standard values for these options are read from session switches when the Debugger is started. The switches have names of the form Debug_*xxx*, where *xxx* is the option name. The standard values indicated below are the default values of these switches. See SJM, Session Switches, for more information on session switches.

Enumerations

`Addresses`

Requests machine information to be put in the displays produced by Stack, Task_Display, Information, and Trace procedures, and to be included when tasks stop in the Debugger. The standard value for this option is false.

`Break_At_Creation`

Causes the equivalent of a breakpoint to be placed at the point where new tasks begin elaboration. The standard value for this option is false.

`Declaration_Display`

Requests all declarations to be displayed when listing source code by means of the Display procedure. The standard value for this option is true.

Delete_Temporary_Breaks

Requests each temporary breakpoint to be deleted once its conditions are met and execution has stopped. The standard value for this option is false.

Display_Creation

Requests a tracelike display of the creation of each task. The standard value for this option is false.

Echo_Commands

Requests that all Debugger commands the user executes be echoed in the Debugger window. The standard value for this option is true.

Freeze_Tasks

Requests that the Debugger attempt to stop all other tasks when a task is stopped by the Debugger. The standard value for this option is false.

Include_Packages

Requests that the Debugger include packages that have completed elaboration in the output generated by the Task_Display procedure. The standard value for this option is false.

Interpret_Control_Words

Causes the Debugger to interpret control stack words when they are displayed using the Memory_Display procedure. The standard value for this option is false.

Kill_Old_Jobs

Causes the last program being debugged to be killed when a new program is started to be debugged. The standard value for this option is true.

Machine_Level

Allows certain machine-level operations. The standard value for this option is false.

No_History_Timestamps

Causes time stamps to be included with each history entry in history displays generated by the History_Display procedure. The standard value of this option is true.

Optimize_Generic_History

Requests that no history be taken in generic instances; in this case, history is taken only for the generic itself, which causes history taking to run considerably faster for generics. The standard value is true.

Permanent_Breakpoints

Specifies whether breakpoints are permanent (must be explicitly deactivated or deleted) or temporary (are deactivated or deleted when the breakpoint first causes execution to stop). The standard value for this option is true.

Put_Locals

Causes local variables to be displayed along with formals when procedure Put is called with locations that are packages or subprograms. The standard value for this option is false.

Qualify_Stack_Names

Causes, when true, the names displayed by the Stack procedure to be fully qualified. When false, the names are the simple names of the subprograms executing in each frame. The standard value for this option is false.

Require_Debug_Off

Requests that the current job being debugged cannot be aborted simply by starting debugging on a new job; debugging must be stopped on the current job by completing its execution or by explicitly executing the Debug_Off procedure to release or abort the job. The standard value for this option is false.

Save_Exceptions

Causes exception-handling information from the Catch and Propagate procedures to be saved from debugging run to debugging run. The standard value for this option is false.

Show_Location

Causes the current source location for the task that stops in the Debugger to be automatically displayed in an Ada window with the location highlighted. The standard value for this option is true. Note that this display will occur only for the control context task, the root task, or (in the case of the Run command) all tasks.

Timestamps

Causes, when true, a time stamp to be displayed with each command and task stop. When false, no time stamp is displayed. The standard value for this option is false.

References

procedure Disable

procedure Enable

type Numeric

---

subtype Path_Name

```
subtype Path_Name is String;
```

Description

Defines a string used to reference declarations, objects, statements, or types within program units.

Used by several operations in this package, this subtype defines the construction of strings that denote specific locations within some program unit. These string names use existing Ada naming rules wherever possible and extend those rules to allow names for anonymous blocks, package and task bodies, accept statement bodies, locations within generic program units, and overloaded names.

Pathnames can be *special names* that indicate that the selection, region, cursor, and image are to be used to specify the location or object. These special names have the form "<SELECTION>" and are described in more detail in LM, Key Concepts. Note that if a special name does not resolve to a location or object (for example, if the special name is "<SELECTION>" and the cursor is not in the selection), the pathname used by the Debugger will be the empty string ("").

Two forms of names are allowed: *full pathnames* and *relative pathnames.* Full pathnames define an absolute location within the program, beginning with a task or library name. Relative pathnames are resolved within the current evaluation or control context, or both, depending on the use of the pathname.

Statements and declarations are numbered. Pathnames to specific statements or declarations include the number of that statement or declaration. Numbering of each group of declarations or statements is independent and begins with 1. When source code is being displayed, these numbers appear at the beginning of statement or declaration lines. In pathnames, statement numbers are differentiated from declaration numbers by a suffix *s* (the default) for statements and *d* for declarations.

Given that the prefix of a pathname refers to a context that contains an object, the following list defines what the next component of a pathname that refers to the object itself would be:

- Library package: Simple package name.
- Library subprogram: Simple subprogram name.
- Library unit generic package: Simple name of the package.
- Library unit generic subprogram: Simple name of the subprogram.
- Library unit generic instantiation: Simple name of the instantiation.
- Package: Simple name of the package.
- Task type: Simple name of the task type.

- Generic instance: Simple name of the instantiation.
- Generic package declaration: Simple name of the generic package.
- Generic subprogram declaration: Simple name of the generic subprogram.
- Subprogram: Simple name of the subprogram.
- Block: Label on the block, if present, or the statement number of the block statement if no label is present (the statement number can be used as a name even if a label is present).
- Accept statement block: Statement number of the accept statement (the accept statement is numbered as a normal Ada statement if it appears alone or as an arm of a select).
- Field of a record: Simple name of the field (this component applies to discriminants, fixed fields, and variant fields).
- Element of an array: Subscript list enclosed in parentheses (the subscripts themselves must be simple constants of the appropriate type).
- The object designated by a pointer: Name "all".

The formal syntax definition of pathnames appears in the following Backus-Naur Form (BNF) definition:

```
Pathname              ::= [Task_Prefix] [Frame_Prefix]
                             ["." Name_Component_Sequence]
                        | Library_Prefix
                             ["." Name_Component_Sequence]
                        | Universe_Prefix
                             [Name_Component_Sequence]
                        | Parent_Prefix
                             [Name_Component_Sequence]
                        | Parent_Library_Prefix
                             [Name_Component_Sequence]
                        | Parent_World_Prefix
                             [Name_Component_Sequence]
                        | Name_Component_Sequence
                        | Special_Name
Task_Prefix           ::= "%" Task_Synonym
Library_Prefix        ::= "." Library_Unit_Name
Frame_Prefix          ::= "_" Frame_Number
Universe_Prefix       ::= "!"
Parent_Prefix         ::= "^"
Parent_Library
  _Prefix             ::= "$"
Parent_World
  _Prefix             ::= "$$"
Name_Component        ::= Component
  _Sequence             | Package_Name ["." Component]
                        | Task_Name ["." Component]
                        | Task_Name ["._" Frame_Number]
                        | Record_Name ["." Component]
                        | Array_Name ["(" Index_List ")"]
                        | Subprogram_Name ["." Component]
                        | Block_Name ["." Component]
                        | Generic_Unit ["." Component]
                        | Generic_Instance ["." Component]
```

```
    Special_Name          ::= "< SELECTION>" "< REGION>" | "< IMAGE>" | "< CUR-
SOR>"
    Index_List            ::= Index ["," Index_List]*
    Index                 ::= Numeric_Literal  |  Identifier
    Package_Name          ::= Name_Component_Sequence
    Task_Name             ::= Name_Component_Sequence
    Record_Name           ::= Name_Component_Sequence
    Array_Name            ::= Name_Component_Sequence
    Subprogram_Name       ::= Name_Component_Sequence
    Block_Name            ::= Name_Component_Sequence
    Component             ::= Identifier ["'" Location_Attribute]
                            | Statement_Number
                            | Declaration_Number
    Statement_Number      ::= Positive ["s"]
    Declaration_Number    ::= Positive "d"
    Positive              ::= Digit+ (non-zero value)
    Location_Attribute    ::= "spec"  | "body"  | "N(" Nickname ")"
    Nickname              ::= Identifier | Positive
    Task_Synonym          ::= Identifier
    Frame_Number          ::= Positive | "-" Positive
    Library_Unit_Name     ::= Identifier
```

In addition to the BNF syntax definition, note the following rules about statement and declaration numbering:

- *Use* clauses and representation specifications are not numbered.
- Parameters (including generic formal parameters) are not numbered.
- Blocks are numbered separately, as are bodies of accepts, and package specs and bodies.
- Statements in exception handlers are numbered continuously with preceding normal statements.
- Accept arms of selects are numbered along with statements at the same level as the enclosing select.
- Frame_Numbers are numbered with respect to the top of the stack (frame 1 is the topmost frame) unless preceded by a minus sign, in which case the frames are relative to the bottom of the stack (frame −1 is the bottommost frame).
- All imported library unit names look through links. The local name of an imported library unit is not known; only the actual name of the imported unit is known. Use the command `Show (Libraries)` to see the library unit names that are known.

These and other constructs are demonstrated in the examples in the Key Concepts for this book and in the following examples.

Example 1

The following are examples of qualified versus unqualified names:

```
%Session_Manager._5.Main_Process.4
```

The name begins with a percent symbol (%) and is qualified. Control and evaluation context are not referenced. A task in the program has called the Set_Task_Name procedure to give itself the name Session_Manager. This name refers to statement 4 of a subprogram named Main_Process that is declared in the subprogram executing in frame 5 of the stack of the task.

```
!Users.Rab.Tests.Regression_Test_12
```

The name begins with an exclamation mark (!) and is fully qualified. Neither control nor evaluation contexts are referenced. The leading exclamation mark implies that the next name (Users) is a library or library unit name contained in the root of the directory system.

The name refers to an object (probably a subprogram) called Regression_Test_12 declared within !Users.Rab.Tests.

```
_2.Condition
```

The leading underscore (_) specifies a reference to stack frame number 2. The task referred to is based on the current control context. If the control context is set to all tasks, then the stack of the last task to stop in the Debugger is referenced. An object named Condition, declared in the subprogram executing in that frame, is designated.

```
User_List.Next.Name
```

No leading special character begins the name. The name is relative to the current evaluation context. Suppose Next and Name are fields of a record and User_List and Next are accesses to that record type. Then this name would refer to field Name of the record pointed to by field Next of the record pointed to by User_List.

If the evaluation context is not set, the top frame of the task specified by the control context (or the last task to stop if the control context is not set) is referenced.

Example 2

The following are examples of naming tasks:

```
%Session_Manager
```

In this example, the name refers to a task that has assigned itself the name Session_Manager. Control and evaluation contexts are not used.

```
%620E
```

In this example, a task number is used to name the task. The task number 620E was obtained from the Task_Display procedure or some other Debugger message that included the task number.

```
.Configurator.Worker
```

The Ada name of a declared (or allocated) task can also be used. Configurator represents a library unit and Worker a single task declared in that unit.

```
_4.Task_Pool(12)
```

An array of tasks, Task_Pool, declared in a subprogram that is executing in frame 4 contains task names. Thus, each element (element 12 in this example) is a task.

Example 3

The following are examples of naming objects relative to a stack:

```
_3.12
```

A numeric suffix on an object name refers to a statement or declaration. In this example, statement 12 of the subprogram executing in frame 3 of the task specified by the control context (the last task to stop if the control context is not set) is named.

```
_1.Foo.12d
```

This name references declaration 12 of subprogram Foo that is declared in the subprogram executing at frame 1 of the task designated by the control context. Note that unless the evaluation context is nonnull, the _1 prefix is used by default.

```
Motor_State.Temperatures.Oil
```

Assume in this example that the evaluation context is null. Motor_State is an object declared in the top frame of the task designated by the control context. The exact interpretations of Temperature and Oil depend on what Motor_State actually is.

Example 4

The following examples illustrate naming array elements.

Consider the following package:

```
package Data_Block is
    type Color is (Red, Blue, Green);

    Info : array (3 .. 35) of integer;

    More_Info : array (1 .. 10, Color, Character)
                                        of Natural;

    Char_Info : array (Character, Character) of integer;
end Data_Block;
```

```
.Data_Block.Info(12)
```

Info names a single-dimensional array. This name references element 12 of the array.

```
.Data_Block.More_Info(1,Blue,'a')
```

In this example, More_Info is a three-dimensional array. The first index type is numeric, the second is an enumeration, and the third is also an enumeration (Character type).

```
.Data_Block.Char_Info(Nul,Bel)
```

Similarly, the two index types of this array are Character type. The indices used are nongraphic literals.

References

procedure Break

procedure Context

procedure Display

procedure Modify

procedure Put

---

procedure Propagate

```
procedure Propagate (Name        : Exception_Name := "<SELECTION>";
                     In_Task     : Task_Name      := "";
                     At_Location : Path_Name      := "");
```

Description

Enters a request to the Debugger that the program being debugged not be stopped when the specified exception is raised in the specified task at the specified location.

By default, the Debugger does not stop any task that raises the selected exception.

The Name parameter names the specific exception or group of exceptions to be ignored by the Debugger. If the Name parameter is the null string (""), or if a special name such as "<SELECTION>" is used but the cursor is not in the selection, the procedure causes execution to proceed when any exception is raised. The reserved string "all" means all exceptions will be ignored. The reserved string "implicit" means exceptions raised implicitly—that is, those raised in the course of the execution of a statement other than raise—will be ignored.

Propagate requests and catch requests combine to determine the action the Debugger takes when an exception is raised.

Note: The Propagate procedure is not the inverse of the Catch procedure. Each of these procedures requests an explicit action for a named exception. The Forget procedure removes catch or propagate requests.

When the Debugger is started, it behaves as though the command `Catch ("all")` has been issued. To make the Debugger ignore exceptions raised in the program, enter the command `Propagate ("all")`.

The In_Task parameter specifies the task in which the exception should be ignored. The reserved string "all" means that the propagate request should apply to all tasks. The null string ("") means that the propagate request should apply to the task specified by the current control context or to all tasks if the control context is not set to a specific task.

The At_Location parameter restricts the location in which the exception is ignored by the Debugger. The null string ("") indicates everywhere in the Environment. If not null, the string specifies a subprogram or statement in which the propagate request applies.

The Debugger maintains a list of catch and propagate requests entered by calls to the Catch and Propagate procedures. When an exception in the user program is raised, the Debugger looks at this list to determine whether to stop the program and inform the user. Catch requests cause the program to stop; propagate requests cause it not to stop.

The action taken in a specific case is determined by the most specific request that applies to the exception. If that request is a catch request, the program stops; otherwise, it does not.

Informally, a catch or propagate request applies to the raising of an exception if the exception name, task name, and location in the exception match the request. More precisely, a request applies to an exception if all of the following are true:

- The name of the exception is the same as the name in the request, the request is for all exceptions, or the request is for implicit exceptions and the exception was raised implicitly.
- The task in which the exception is raised equals the task in the request or the request is for all tasks.
- The exception is raised in a statement or declaration that the request specified, the point of raise is in a subprogram that the request specified, or the request is for all locations.

If more than one catch or propagate request applies to a specific raising of an exception in a program, the more specific one determines the action the Debugger will take.

Requests are considered more specific if they specify a smaller number of cases. Thus, the more parameters of the request that are specified, the more specific the request. More formally:

- A request that specifies a subprogram and statement is more specific than one that specifies only a subprogram.
- A request that specifies a subprogram is more specific than one that specifies all locations.
- A request that specifies a task is more specific than one that does not.
- A request that names an exception is more specific than one that specifies implicit or all exceptions.
- A request that specifies implicit exceptions is more specific that one that specifies all exceptions.

These rules are applied in combination in the order given. The location is a stronger specification than the task, and the task restriction is a stronger specification than the exception name.

The command `Propagate ("all", "", "")` is the least specific and causes the Debugger not to stop program execution for any exceptions unless a more specific catch request applies to it.

To remove a catch or propagate request, use the Forget procedure.

If a propagate request has parameters that exactly match a previous catch request, the catch request is removed first.

The command Show (Exceptions) displays all catch and propagate requests ordered, from most specific to least specific. For each request, the exception name, location, and task restrictions are listed.

Parameters

Name : Exception_Name := "<SELECTION>";

Specifies the exception for the propagate request. By default, the selected exception will be propagated.

If the Name parameter is the null string (""), or if a special name such as "<SELECTION>" is used but the cursor is not in the selection, the procedure propagates for all exceptions.

The reserved name "all" stands for all exceptions.

The reserved name "implicit" stands for all exceptions raised implicitly—that is, those raised by some language construct other than a raise statement. These implicit exceptions include only predefined language exceptions such as Constraint_Error and Tasking_Error.

If the exception name is not fully qualified (and nonnull), it is interpreted relative to the current evaluation context.

In_Task : Task_Name := "";

Specifies the task that should be monitored for the exception specified by the Name parameter. If null, the task specified by the control context is used. If the control context is not set to a specific task, the propagate request applies to all tasks. The reserved name "all" also implies all tasks. By default, all tasks should be monitored unless the control context has been set.

If the task name is not fully qualified, not null, and not a simple task name—beginning with a percent symbol (%) or a digit—the name is interpreted relative to the current evaluation context.

```
At_Location : Path_Name := "";
```

Specifies a location restriction for the exception propagate request. The raising of the specified exception requests the specified task not to stop only if the point of raise of the exception is in the subprogram or at the statement specified by the At_Location parameter. By default, there are no location restrictions and the exception is ignored wherever it is raised.

If the At_Location parameter specifies an Ada unit, the exception is propagated only inside that unit; it is not propagated inside any Ada units nested inside it, including nested blocks and accept statements. Separate requests have to be entered if this effect is desired.

If At_Location specifies a particular statement or declaration, the exception is propagated only when it is raised in the statement or declaration.

If At_Location is null, the propagate request applies anywhere.

Note: The evaluation context is used in interpreting At_Location only if it is not null. If At_Location is null, the propagate request applies throughout the program, independent of the evaluation context.

If not fully qualified and not null, the Path_Name parameter is interpreted relative to the current evaluation context.

Restrictions

A maximum of 40 requests to catch or propagate exceptions can be set.

References

procedure Catch

procedure Forget

procedure Show (Exceptions)

---

procedure Put

```
procedure Put (Variable    : Path_Name := "<SELECTION>";
               Stack_Frame : Integer   := 0);
```

Description

Displays the value of the specified object in the Debugger window with formatting based on the type of the object.

By default, the Put procedure displays the value of the selected object in the most recent frame of the stack of the last task to stop in the Debugger. If the object is declared in a library unit package, the value is determined from the package's state, not from a stack frame.

The Variable parameter is the primary means for specifying the object for which to display the value. The Stack_Frame parameter provides a convenient means for specifying either the context for interpretation of the Variable parameter or the actual values to be displayed. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Put command is bound to a key. If the Stack_Frame parameter is nonzero and the Variable parameter specifies a relative pathname, the actual pathname used to specify the object to display is composed by appending the string *"_n"* to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Variable parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If selection is used to specify the object, the stack is searched from the frame indicated by the value of the Stack_Start numeric option—by default, the top frame—to the bottom frame to find the first occurrence of the object or the subprogram selected. Note that if the Stack_Frame parameter is nonzero, this search will begin in the frame indicated by that parameter instead. This search is limited to the number of frames of the stack indicated by the Stack_Count numeric option—by default, 10 frames. If selection is not used, this searching is not performed.

If the Variable parameter is the null string (""), or if a special name that does not resolve to an object is used (for example, if the Variable parameter is "<SELECTION>" and the cursor is not in the selection), the object designated by the current evaluation context is displayed. If the evaluation context is not set (or refers to a stack frame), all parameters to the subprogram in execution in that stack frame are displayed (if the Put_Locals option is true, the values of the local variables for the subprogram are also displayed). Unless the control context is explicitly set, the top frame of the last task to stop in the Debugger is displayed. Note that if selection is used and if the statement executing in the specified stack frame is an assignment or return statement, the value of the statement's lefthand side or the return value, respectively, is displayed.

The pathname given is interpreted according to the naming rules described under the Path_Name subtype in this section. If the name is unqualified, the current evaluation context determines the context in which the pathname is interpreted. If the current evaluation is not explicitly set, the context for interpretation defaults to the control context. If the control context is not set, it defaults to the top stack frame of the last task reported stopped by the Debugger.

If a scalar is specified, that scalar is displayed.

If a structure (record or array) is specified, the display is controlled by several option flags. In particular, the Display_Level (Numeric type) flag determines the number of nesting levels of the object to be displayed. The Pointer_Level flag determines the number of levels of pointers in the object to be expanded.

The Element_Count and First_Element flags control the number of array elements and which element is displayed first, respectively.

If the array displayed is a string, then up to 73 characters are displayed, independent of the Element_Count flag. Strings are specially formatted and appear in quotes. Control characters appear in inverse video, except for the linefeed character, which breaks the line.

These flags can be set using the Set_Value procedure.

The Debugger provides a facility with which users can create *special display* routines for displaying data values. This facility can be useful if the user does not want the Debugger to use the structural information to display the value, but instead wants to supply a more useful or more efficient display procedure.

Users can provide display procedures for the type and register them with the Debugger using the !Tools.Debug_Tools.Register generic procedure (typically in the Debugger_Initialization procedure). See the documentation of the Register procedure in package Debug_Tools for more information on using special displays.

When special displays are registered for a type, the Put command causes that function to be invoked to form an image for the value of the object of the type. Such an image is enclosed in braces ({}) to distinguish it from the standard output of the Put command.

Parameters

```
Variable :  Path_Name := "<SELECTION>";
```

Specifies the object to be displayed. See the description above for more information on how this parameter is interpreted. By default, the selected object is displayed.

```
Stack_Frame :  Integer := 0;
```

Specifies the stack frame containing the value of the object to display, or specifies the stack frame to display. Note that the stack frame can also be specified in the Variable parameter; this parameter typically is used when its value is provided using argument prefix keys when the Put command is bound to a key. If the Stack_Frame parameter is nonzero and the Variable parameter specifies a relative pathname, the actual pathname used to specify the object to display is composed by appending the string *"_n"* to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Variable parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

See the description above for more information on how this parameter is interpreted.

Restrictions

The parameter must specify an object that has a value.

A specific package, task, or subprogram frame must be referenced by the pathname. If a selection is not being used, a reference to a subprogram name is illegal because the Debugger does not know in which activation of the subprogram to look. The same applies to generic packages and task types; a specific instance must be referenced.

Only single-dimensional arrays can be displayed. Individual elements of higher-dimensional arrays can be displayed, but not the entire array in a single command.

Errors

The parameter specifies an object that has no value—for example, a field of a variant record that is not present because of the value of the discriminant.

There may be many problems with the name. Refer to the naming rules discussed under the Path_Name subtype in this section and to the examples in the Key Concepts.

A value can be uninitialized.

Note: Uninitialized scalar variables can be detected by the Environment. However, the Environment does not know whether fields of structures are initialized; therefore, the Debugger reports a random value if an uninitialized field of a structure is displayed.

The name can include a subscript that is out of range for an array being indexed.

The name can dereference a pointer that is null.

Other errors that occur usually result from attempting to access objects that have not been elaborated.

Example

A sample Debugger session that includes use of the Put and Display procedures is shown below:

```
Beginning to debug: DEBUG_DOC.TEST_PROGRAMS.EXAMPLE_FOR_MANUAL'BODY'V(1)  %
    !USERS.BLB.DEBUG_DOC.TEST_PROGRAMS.EXAMPLE_FOR_MANUAL
Stop at: .command_procedure,  Root task: [Task : ROOT_TASK, #1134E1].

Execute ("all");
User break: .EXAMPLE_FOR_MANUAL.1s  [Task : ROOT_TASK, #1134E1].

Display ("%ROOT_TASK._1", Ø);
      procedure .EXAMPLE_FOR_MANUAL  is
    1       type TERMINAL is (VT1ØØ, DASHER, CIT5ØØ, ADM3);
    2       type ID is new INTEGER;
    3       type TERM_ARRAY is array (1 .. 3) of TERMINAL;
    4       type R1 is record
                F1 : INTEGER;
                F2 : CHARACTER;
                F3 : TERM_ARRAY;
            end record;
    5       type AR1 is access R1;
    6       B1 : TERMINAL := DASHER;
    7       B2 : ID := ID'FIRST;
```

```
    8       B3 : R1 := R1'(F1 => Ø, F2 => 'b', F3 => (VT1ØØ, DASHER, VT1ØØ));
    9       B4 : AR1 := new R1'(F1 => -2Ø, F2 => ASCII.ESC, F3 => (DASHER, ADM3,
             CIT5ØØ));
        begin
*   1       DEBUG_TOOLS.USER_BREAK ("");
        end;
```

Now, the values of the variables are displayed. Note that the default context is the top frame of the last task to stop—in this case, the procedure frame containing the variables of interest.

Before the first Put command is executed, the variable name B1 is selected. The Debugger echoes the following when [Put] is pressed:

```
Put ("%ROOT_TASK._1.B1");
DASHER
```

Note that, although the Value parameter to the Put command specified that a selection be used, the name of the selected object is echoed in the command.

The following Put commands had string parameters of the form "b1" for their Value parameters. Note that selection could have been used as well:

```
Put ("%ROOT_TASK._1.b1");
DASHER

Put ("%ROOT_TASK._1.b2");
-2147483647

Put ("%ROOT_TASK._1.b3");

[ F1 => Ø
  F2 => 'b'
  F3 =>
  [ 1 .. 3 ]
  [ 1 => VT1ØØ
    2 => DASHER
    3 => VT1ØØ ]
 ]

Put ("%ROOT_TASK._1.b4");
#1134E1 #8Ø -->
  [ F1 => -2Ø
    F2 => ASCII.ESC
    F3 =>
    [ 1 .. 3 ]
    [ 1 => DASHER
      2 => ADM3
      3 => CIT5ØØ ]
   ]

Put ("%ROOT_TASK._1.b4.f3(2)");
ADM3

Put ("%ROOT_TASK._1.b4.f3(5)");
```

```
Display error:
Array index out of bounds

Display ("%ROOT_TASK._1.b4", Ø);
        access R1

Display ("%ROOT_TASK._1.b2", Ø);
        range -2147483647 .. 2147483647
```

References

procedure Context

type Numeric

type Option

subtype Path_Name

procedure Set_Value

procedure Debug_Tools.Register

procedure Debug_Tools.Un_Register

---

procedure Release

```
procedure Release (Name : Task_Name := "");
```

Description

Releases a task (or tasks) from the held state and moves the task to the stopped state.

A task that is held either must be explicitly allowed to execute or can be moved to the stop state by means of this operation. After the Release procedure is executed, the referenced task will still be stopped as though a Stop procedure had been executed.

The command `Release ("all")` clears any hold conditions. The command `Execute ("all")` then clears any stop conditions.

Parameters

`Name : Task_Name := "";`

Specifies the task to be released. The default is the task specified by the control context or all tasks if the control context is not explicitly set.

The string "all" can be used to specify that all tasks are to be released.

References

procedure Execute

procedure Hold

procedure Stop

procedure Remove

```
procedure Remove (Breakpoint : Natural;
                  Delete     : Boolean := False);
```

Description

Deactivates and possibly deletes the specified breakpoint(s).

The specified breakpoint will no longer interrupt execution. The breakpoint, if not deleted, remains defined and can be reactivated at some later time by using the Activate procedure. If deleted, the breakpoint no longer exists and cannot be reactivated.

Parameters

`Breakpoint : Natural;`

Specifies which breakpoint to be deactivated. The number is assigned when the breakpoint is created.

The number 0 is used to represent all breakpoints. Thus, the command `Remove(0, true)` deletes all breakpoints.

`Delete : Boolean := False;`

Specifies whether to delete the breakpoint. The default is not to delete the breakpoint.

Errors

The breakpoint does not exist.

References

procedure Activate

procedure Break

procedure Show (Breakpoints)

procedure Reset_Defaults

```
procedure Reset_Defaults;
```

Description

Resets all flag values, numeric values, and Boolean options to their standard values and unregisters all special displays.

See procedure Debug_Tools.Register for more information on special displays.

If this procedure is executed from your login procedure, it automatically starts the Debugger so that you do not have to wait for it to start when you debug the first program in your session. The call to Reset_Defaults does not bring the Debugger window onto your screen.

procedure Run

```
procedure Run (Stop_At : Stop_Event := Debug.Statement;
               Count   : Positive   := 1;
               In_Task : Task_Name  := "");
```

Description

Executes the specified task(s) until the stop event has occurred the number of times specified by the Count parameter.

By default, the last task to stop in the Debugger is run.

The Run procedure causes the specified task(s) to begin execution and then stop after a specific situation described by the Stop_At parameter has occurred Count times. A message is then displayed in the Debugger window, and the task stops.

If the Freeze_Tasks flag is true and all tasks are stopped implicitly as a result of an individual task being stopped, the Run procedure commences execution of these implicitly stopped tasks.

The task(s) may stop before Count events have occurred for some other reason. For example, the task may raise an exception that causes the Debugger to stop its execution or reach a breakpoint. If this happens, the stepping operation is still in progress. When the task's execution is resumed by the user, the task still stops when Count events finally occur.

The Clear_Stepping procedure can be used to clear the stepping condition before it has been triggered.

The most common use of the Run procedure is for the stepping of a task statement by statement. The Run procedure with no parameters does this. After each execution, the Debugger reports the location of the next statement to be executed by the task.

If the analysis of a single subprogram is desired, a breakpoint can be set at the beginning of the procedure and, once the task reaches the breakpoint, the command `Run (Local_Statement)` can be given to execute statements within that procedure. The task being stepped in this way always stops within this subprogram. Calls are executed as part of a single step when doing Local_Statement stepping. If the subprogram returns, the task stops at the beginning of the next statement it is about to execute.

Stepping substantially slows execution of the program. It may be wise to set breakpoints near the region of interest in the program and step through only that part.

Parameters

```
Stop_At : Stop_Event := Debug.Statement;
```

Specifies the event that will cause the task to stop. The default is any statement.

```
Count : Positive := 1;
```

Specifies the number of times the event must be executed before the task stops. The default is the first execution of the event, which stops the task.

```
In_Task : Task_Name := "";
```

Specifies the task to be run. The default is the task specified by the control context. If the control context is not explicitly set, the default is the last task to stop in the Debugger. The string "all" can be used to specify that all tasks are to be run.

Restrictions

Only one stepping operation can be in progress for a task at any given time. Starting a new one cancels an existing operation for a given task.

Errors

The number of debugging operations that can be applied to a specific task is limited. If too many breakpoints are set in a given task, it may not be possible to execute the Run procedure for that task.

Example

When the Debugger stops and displays:

```
Break 1: .STEPPING_TEST.2s  [#620E]
```

the user enters the command:

```
Run (Statement, 3, "%602E");
```

and the Debugger then displays:

```
Step: .STEPPING_TEST.5s  [#620E]
```

References

procedure Clear_Stepping

procedure Show (Steps)

---

procedure Set_Task_Name

```
procedure Set_Task_Name (For_Task : Task_Name := "";
                         To_Name  : String    := "");
```

Description

Assigns a string *nickname* for the named task.

The name can be used by the user in various Debugger commands as a synonym for the task. The nickname can also be set by the Debug_Tools.Set_Task_Name procedure.

To use such a nickname in a command, the name must be preceded by a percent symbol (%). The string passed to this procedure, however, should not have a leading percent symbol.

It is good practice to call the Set_Task_Name procedure in important (if not all) tasks to identify them easily during debugging. The call is especially important when multiple instances of the same task are created. It is some extra work to give each a unique name, but the effort often greatly simplifies the task of understanding what is going on during debugging.

If the name passed to the Set_Task_Name procedure has already been used by a different task, then the name is removed from that old task and assigned to the present caller to the Set_Task_Name procedure. Thus, only one task has a given name at a given time.

The name `Root_Task` is automatically assigned to the root task (the command task) in a job. To avoid confusion, it is best not to reassign this name.

Parameters

`For_Task : Task_Name := "";`

Specifies the task for which a nickname is desired. By default, when For_Task is the null string (“”), the task last stopped in the Debugger is used unless the control context is set. If the control context is set, its value is used when For_Task is the null string.

```
To_Name :  String := "";
```

Specifies the name to be assigned to the task. It is best to limit the name to 40 characters, because Debugger displays that include the task name are not as readable if excessively long names are used.

The string must be a legal Ada identifier. If the string contains illegal characters an error message appears.

References

type Path_Name

procedure Task_Display

procedure Debug_Tools.Set_Task_Name

---

procedure Set_Value

```
procedure Set_Value (Variable : Numeric;
                     To_Value : Integer);
```

Description

Sets the numeric variable flag controlling the behavior of the Debugger to the specified value.

See the Numeric type for more information on the numeric variable flags and their meanings. See the Option type for more information on the Boolean option flags and their meanings.

Parameters

`Variable : Numeric;`
Specifies the numeric variable to be changed.

`To_Value : Integer;`
Specifies the value to which the variable is set.

References

type Numeric

type Option

procedure Show

```
procedure Show (Values_For : State_Type := Debug.Breakpoints);
```

Description

Displays information about various Debugger facilities.

By default, the list of breakpoints is displayed.

If the Values_For parameter has the value of All_State, it causes all the information to be displayed. Other values cause only the state for specific facilities to be displayed. The content of each of those displays is described in examples 1 through 9, below.

Parameters

```
Values_For : State_Type := Debug.Breakpoints;
```

Specifies what class of state to be displayed. The default is the class of all breakpoints.

Example 1

The command:

```
Show (Breakpoints);
```

produces the display:

```
Active Permanent Break 4 at !USERS.JACK.TEMP.3d [any task]
Inactive Permanent Break 5 at !USERS.JACK.T.1s [any task]
Active Temporary Break 6 at !USERS.JACK.JILL.1s [task : #2D110]
```

The display of breakpoint information lists all existent breakpoints. Listed for each breakpoint is information about whether the breakpoint is active or inactive, permanent or temporary, the break number, the location in which the break is set, and the task to which the break applies.

Example 2

The command:

```
Show (Contexts);
```

displays the current control and evaluation contexts.

The commands:

```
Context (Control, "%2D11Ø");
Context (Evaluation, "!USERS.PHIL.PIE_CHART");
Show (Contexts);
```

produce the display:

```
Evaluation context: !users.phil.pie_chart
Control context: #2D11Ø
```

The commands:

```
Context (Control, "all");
Context (Evaluation, "");
Show (Contexts);
```

produce the display:

```
Evaluation context:
Control context: all
```

Example 3

The command:

```
Show (Flags);
```

displays the name of each known flag and its current value:

```
DISPLAY_COUNT                    1Ø
DISPLAY_LEVEL                    3
ELEMENT_COUNT                    25
FIRST_ELEMENT                    Ø
HISTORY_COUNT                    1Ø
HISTORY_ENTRIES                  1ØØØ
HISTORY_START                    1Ø
MEMORY_COUNT                     3
POINTER_LEVEL                    3
STACK_COUNT                      1Ø
STACK_START                      1
ADDRESSES                       FALSE
BREAK_AT_CREATION               FALSE
DECLARATION_DISPLAY             TRUE
DELETE_TEMPORARY_BREAKS         FALSE
DISPLAY_CREATION                FALSE
ECHO_COMMANDS                   TRUE
FREEZE_TASKS                    FALSE
INCLUDE_PACKAGES                FALSE
INTERPRET_CONTROL_WORDS         TRUE
KILL_OLD_JOBS                   TRUE
MACHINE_LEVEL                   FALSE
NO_HISTORY_TIMESTAMPS           TRUE
OPTIMIZE_GENERIC_HISTORY        TRUE
PERMANENT_BREAKPOINTS           TRUE
PUT_LOCALS                      FALSE
QUALIFY_STACK_NAMES             FALSE
REQUIRE_DEBUG_OFF               FALSE
```

```
SAVE_EXCEPTIONS                 FALSE
SHOW_LOCATION                   TRUE
TIMESTAMPS                      FALSE
CACHE_STACK_FRAMES              TRUE
```

Example 4

The command:

```
Show (Exceptions);
```

displays all active exception requests. At least one request is always active; it indicates what to do for exceptions not covered by other exception requests.

The exception requests are listed from the most specific to the least specific. The most specific requests are applied first to determine what action to take when an exception is raised. The following example shows how, after exception-handling requests are set up, the display changes to reflect those requests:

```
Show (EXCEPTIONS);
    Debugger stops on: unlisted exceptions

Catch ("constraint_error", "all", "");
The exception constraint_error will be caught when raised in any
    location in all tasks.

Propagate ("tasking_error", "all", "");
The exception tasking_error will be propagated when raised in any
    location in all tasks.

Catch ("implicit", "all", "");
Implicit exceptions will be caught when raised in any location in
    all tasks.

Show (EXCEPTIONS);
    Debugger stops on: Constraint_Error
    Debugger doesn't stop on: Tasking_Error
    Debugger stops on: implicit
    Debugger stops on: unlisted exceptions

Forget ("constraint_error", "all", "");
The exception constraint_error has been forgotten for all locations
    in all tasks.

Propagate ("", "all", "");
Unlisted exceptions in unlisted locations and tasks will be propagated.

Catch ("!io.io_exceptions.layout_error", "1cf8e1", "");
The exception !io.io_exceptions.layout_error will be caught when
    raised in any location in task #1CF8E1.

Show (EXCEPTIONS);
    Debugger stops on: !Io.Io_Exceptions.Layout_Error when raised in task
    #1CF8E1
    Debugger doesn't stop on: Tasking_Error
    Debugger stops on: implicit
    Debugger doesn't stop on: unlisted exceptions
```

Example 5

The command:

```
Show (Libraries);
```

example displays a list of current history being taken. lists the names of the currently registered libraries and their library units.

If Addresses is enabled, other machine information about the library is also displayed.

```
Show (LIBRARIES);
Libraries in use by this program:
  Library: TEST_LIBRARY
  Main Unit: M1
  Library Units:
    package TEXT_IO
    procedure TEST
    procedure MAIN
```

Example 6

The following example lists information about trace conditions that are enabled. The name of each affected task and the type of tracing enabled for it are displayed:

```
Trace (TRUE, STATEMENT, "all", "");
Statement tracing has been enabled for all locations in all tasks.

Trace (TRUE, CALL, "%1cf8e1", "");
Call tracing has been enabled for all locations in
task A, #1CF8E1.

Show (TRACES);
Tasks which are tracing calls:
  #1CF8E1: at all locations
Tasks which are tracing statements:
  all: at all locations
No tasks are tracing exceptions.
```

Example 7

This example displays a list of current history being taken:

```
Take_History (TRUE, CALL, "all", "");
Call history-taking has been enabled for all locations in all tasks.

Take_History (TRUE, STATEMENT, "all", "");
Statement history-taking has been enabled for all locations in all tasks.

Take_History (TRUE, EXCEPTION_RAISED, "all", "");
Exception history-taking has been enabled for all locations in all tasks.

Show (HISTORIES);
History of Calls is being recorded for:
  all tasks at all locations
```

```
    History of Statements is being recorded for:
      all tasks at all locations
    History of Exceptions is being recorded for:
      all tasks at all locations
```

Example 8

This example displays a list of currently active stop and hold requests. If stop or hold requests for all tasks have been entered, and later some tasks are released or started, then the tasks that are exempt from the stop or hold request are listed.

```
    Hold ("all");
    Each task will stop by its next statement.

    Release ("%1cf8e1");
    Fine.

    Stop ("%1cf8e1");
    Task #1CF8E1 will stop by its next statement.

    Show (STOPS_AND_HOLDS);
    Stops and Holds:
All tasks:  Hold
#1CF8E1 Stop; exempt from hold all
```

Example 9

The following example lists the names of tasks that currently have stepping operations in progress. A stepping operation is initiated by the Run procedure and is terminated when the stepping operation completes, a new one is initiated, or the Clear_Stepping procedure is executed for a task.

```
    Show (Steps);
    Task #1CF8E1 stepping statements.
```

Example 10

Other information that can be displayed includes Statistics (State_Type type). Repeated use of the command `Show (Statistics)` can give you an idea whether any Debugger interactions are occurring within your program. Look for changing numbers.

procedure Show
package !Commands.Debug

References

procedure Break

procedure Catch

procedure Context

procedure Enable

procedure Flag

procedure Forget

procedure Hold

procedure Propagate

procedure Run

procedure Set_Value

type State_Type

procedure Stop

procedure Trace

procedure Debug_Tools.Register

---

procedure Source

```
procedure Source (Location     : Path_Name  := "<SELECTION>"
                  Stack_Frame  : Integer    := 0);
```

Description

Finds the source for the specified location and displays that location highlighted in an Ada window.

By default, the source location for the next statement to be executed by the last task stopped in the Debugger is highlighted in an Ada window.

The Location parameter is the primary means for specifying the source to display. The Stack_Frame parameter provides a convenient means for specifying a frame to display. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Source command is bound to a key. If the Stack_Frame parameter is nonzero and the Location parameter specifies a special name (such as "<SELECTION>"), the Location parameter is ignored and the source for the frame specified by Stack_Frame is displayed. If the Stack_Frame parameter is nonzero and the Location parameter specifies a relative pathname, the actual pathname used to specify the object to display is composed by appending the string *"_n"* to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the Location parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If the null pathname ("") is specified, or if a special name (such as "<SELECTION>") is used but the item is not designated in the Debugger window, then the procedure finds the location in which the current task last stopped. Otherwise, the procedure attempts to find the most reasonable definition of the given pathname and the current debugging context.

If the named location is:

- A task name, the corresponding task declaration is viewed.
- A frame number, the corresponding subprogram executing on that stack frame is viewed.
- A library unit, the library unit is viewed.
- A statement or declaration, the containing unit is viewed and the statement or declaration in the unit is highlighted.
- Any other object, the object or its type is viewed.

Parameters

```
Location : Path_Name := "<SELECTION>";
```

Specifies the location to be viewed. The interpretation of this parameter is discussed in more detail in the description above. By default, the source for the next statement to be executed by the last task to stop in the Debugger is highlighted in an Ada window.

```
Stack_Frame : Integer := 0;
```

Specifies the frame for which to display the source. The interpretation of this parameter is discussed in more detail in the description above. By default, this parameter is ignored and the item specified by the Location parameter is displayed.

References

procedure Display

procedure Stack

```
procedure Stack (For_Task : Task_Name := "";
                 Start    : Integer   := Ø;
                 Count    : Natural   := Ø);
```

Description

Displays the specified frames of the stack of the named task.

By default, the stack for the task last stopped in the Debugger is displayed.

The Start parameter specifies the number of the first frame to be displayed, and the Count parameter specifies the number of frames to be displayed.

The stack of any task in the job being debugged can be displayed. If the task is running at the time the Stack procedure is called, the task is temporarily stopped so that the display shows a consistent set of frames. After the display, the execution of the task continues.

If you plan to take some action based on information in the display of the stack, such as displaying the source for a frame, be sure that the task is stopped. Otherwise, the information in the display probably will be obsolete by the time the later command is issued.

Several options control the information displayed in the stack display. Qualify-_Stack_Names (Option type), when true, causes the Ada names listed in the frame to be fully qualified. If false, the standard setting, only the simple name of each subprogram is listed.

The Addresses option, when true, causes machine information to be included in the display. The specific information includes the value of the program counter for that frame (segment and offset), the value of the stack frame pointer for that frame, the outer frame pointer for that frame (points to the declarative context of the subprogram, a control stack address), and the lexical level of the subprogram.

Parameters

```
For_Task : Task_Name := "";
```

Specifies the name of the task whose stack is to be displayed. The default is the stack of the task specified by the current control context if the context is set, or it is the last task stopped if the context is not set.

```
Start : Integer := 0;
```

Specifies the starting frame to be displayed. Stack frames are numbered from the top (most recently called subprogram) starting with 1. If the value of Start is negative, the frame is referenced relative to the bottom of the stack. Thus, −1 is the bottommost frame, −2 the frame above that, and so on.

The default value for the Start parameter is controlled by Stack_Start (Numeric type). The standard value is 1.

```
Count : Natural := 0;
```

Specifies the maximum number of frames to be displayed. The display always stops if no more frames remain to be displayed.

The default value for the Count parameter is controlled by the Stack_Count option. The standard value is 10.

Errors

The task name may be invalid.

The area of the stack to be displayed may be completely outside the current area of the stack.

Example

Assume that the following program is being debugged and has stopped in the breakpoint generated by the call to Debug_Tools.User_Break in procedure A:

```
with Debug_Tools;
procedure Nested_Calls is
    procedure A is
    begin
        Debug_Tools.User_Break ("stopped in a");
    end A;
    procedure B is
    begin
        A;
    end B;
    procedure C is
    begin
        B;
    end C;
    procedure D is
    begin
        C;
    end D;
begin
```

```
    D;
end Nested_Calls;
```

If [Stack] is pressed, the stack is displayed in the Debugger window:

```
Stack ("%ROOT_TASK", Ø, Ø);
Stack of task ROOT_TASK, #340EØ:
_1: A.1s
_2: B.1s
_3: C.1s
_4: D.1s
_5: NESTED_CALLS.1s
_6: command_procedure.1s
_7: command_procedure     [library elaboration block]
```

The first line tells you the task whose stack is being displayed. Here the root task of the main program whose string name is Root_Task and whose task number is #340E0 is having its stack displayed.

The subsequent lines display the frames of the stack for the task. Each frame is prefixed with its number (for example, _1:) and contains the location in the source program corresponding to that frame. The top frame, frame 1, is the most recent frame. Earlier frames have higher numbers. In this case, the bottom frame, number 7, is the code that elaborated the command that called the Nested_Calls program.

If the Qualify_Stack_Names option is set to true (with the Enable command), the stack frames will be displayed with the full pathnames for their source locations. For example:

```
Enable (QUALIFY_STACK_NAMES, TRUE);
The QUALIFY_STACK_NAMES flag has been set to TRUE.

Stack ("%ROOT_TASK", Ø, Ø);
Stack of task ROOT_TASK, #340EØ:
_1: .NESTED_CALLS.A.1s
_2: .NESTED_CALLS.B.1s
_3: .NESTED_CALLS.C.1s
_4: .NESTED_CALLS.D.1s
_5: .NESTED_CALLS.1s
_6: .command_procedure.1s
_7: .command_procedure     [library elaboration block]
```

References

type Numeric

type Option

---

type State_Type

```
type State_Type is (All_State, Breakpoints, Contexts, Exceptions, Flags,
                    Histories, Libraries, Special_Types, Steps,
                    Stops_And_Holds, Traces, Active_Items,
                    Exception_Cache, Inner_State, Statistics);
```

Description

Defines the Debugger state information that can be displayed by the Show procedure.

Enumerations

`All_State`

Specifies that the Breakpoints through Stops_And_Holds enumerations are displayed.

`Breakpoints`

Specifies that all defined (active and inactive) breakpoints are displayed.

`Contexts`

Specifies that the control and evaluation contexts are displayed.

`Exceptions`

Specifies that the catch and propagate exception requests are displayed.

`Flags`

Specifies that flag values in the Debugger, including those in the Option and Numeric types, and the string-named flags set in the Flag procedure are displayed.

`Histories`

Specifies that the current requests for history taking are displayed.

`Libraries`

Specifies that all active libraries are displayed.

`Special_Types`

Specifies that the current set of registered special displays are to be displayed. See the Debug_Tools.Register procedure for more information on special displays.

Steps

Specifies that all tasks that are stepping and their current state are displayed.

Stops_And_Holds

Specifies that the current stop and hold requests are displayed.

Traces

Specifies that information is displayed about which tasks have tracing enabled and what task types they are.

Active_Items

For the use of Rational technical representatives only.

Exception_Cache

For the use of Rational technical representatives only.

Inner_State

For the use of Rational technical representatives only.

Statistics

For the use of Rational technical representatives only.

References

procedure Show

procedure Debug_Tools.Register

---

procedure Stop

```
procedure Stop (Name : Task_Name := "");
```

Description

Stops execution of the specified task(s).

The task is stopped at the beginning of the next executing statement in that task. This procedure allows for rendezvous and complex statements to complete before the stop takes effect. A message is displayed for each task when it stops. If the task was previously stopped or held, the procedure has no effect.

If the Freeze_Tasks flag is true, the Debugger attempts to stop all other tasks. Note that in this mode other tasks may not actually stop because they may be in rendezvous with the stopped tasks, waiting for an entry call, and so on.

Stopping a task means that the task can be commanded to execute again either implicitly or explicitly. The task then can be explicitly named in an Execute or Run procedure or can be implicitly allowed to run as part of a group or set of tasks allowed to run. This procedure contrasts with the , which requires explicit commands to commence execution; that is, the task must be explicitly named to be allowed to run.

The reserved name "all" implies that all tasks are to be stopped. The null string ("") implicitly specifies the name "all" unless the control context is set, in which case it specifies the control context task. This includes tasks that are created after the command `Stop ("all")` takes effect. The default for the task name is the current control context. If that context is null, the default is all tasks.

Individual tasks can be stopped and allowed to continue. For example, all tasks can be stopped and then some allowed to continue by means of the Execute procedure. If the continued tasks had created new tasks, the new tasks would stop because the command `Stop ("all")` would still be in effect.

The command `Execute ("all")` clears the command `Stop ("all")`. Starting individual tasks does not clear this stop condition.

Parameters

`Name : Task_Name := "";`

Specifies the task(s) to be stopped. The default is the current control context task. If the control context is not set, the default is all tasks.

Restrictions

The named task must exist.

Example

The following example describes the sequence of events that occurs after a Stop command is given when debugging a multitasking program:

```
Each task will stop by its next statement.
Stop:   .PRODUCER_CONSUMER.CONSUMER.4S   [Task : CONSUMER, #2F8D4].
Stop:   .PRODUCER_CONSUMER.PRODUCER.4S   [Task : PRODUCER, #2F4D4].
Stop:   .PRODUCER_CONSUMER.QUEUE.7S   [Task : QUEUE, #2FØD4].
```

The first line appears immediately after the Stop command is given. Then each task stops one by one. There may be some delay because each task must complete its current statement or declaration before being stopped by the Debugger. If there are delay statements, rendezvous, or complex statements, the task may require some time before being stopped.

References

procedure Execute

procedure Hold

procedure Release

procedure Run

procedure Task_Display

type Stop_Event

```
type Stop_Event is (About_To_Return,  Begin_Rendezvous,  End_Rendezvous,
                    Local_Statement,  Machine_Instruction,
                    Procedure_Entry,  Returned,  Statement);
```

Description

Defines the events used to specify when a task should stop.

Used in the , the Stop_Event type selects what event in the execution of a task will cause that task to stop.

Enumerations

About_To_Return

Specifies that the task is about to return from a subprogram. This enumeration is a return statement or the end of a subprogram.

Begin_Rendezvous

Specifies that the task is starting to rendezvous with another task. This enumeration applies only to the accepting task. Entry calls do not cause tasks that are stepping to break until a rendezvous is begun.

End_Rendezvous

Specifies that the task is about to complete a rendezvous with another task. This enumeration applies only to the accepting task and occurs after the last statement that is part of the rendezvous is executed.

Local_Statement

Specifies that the task completed execution of the current statement. Local_Statement treats subprograms called from the current statement as part of it.

Machine_Instruction

Specifies that the task has completed execution of the current machine instruction. This event is not currently implemented for the R1000 target.

Procedure_Entry

Specifies that the task has just entered a subprogram. The task stops before elaboration of the first declaration or, if no declarations exist, before execution of the first statement.

Returned

Specifies that the task has just returned from the current subprogram and is about to execute the next statement. Note that many levels of returns may occur if a number of subprograms are at their last statement.

Statement

Specifies that the task has reached the beginning of the next statement. It may not have completed the current statement if that statement is, or includes, a subprogram call. In that case, the task stops at the first statement of declaration of the called subprogram.

---

procedure Take_History

```
procedure Take_History (On          : Boolean     := True;
                        Event       : Trace_Event := Debug.All_Events;
                        For_Task    : Task_Name   := "";
                        At_Location : Path_Name   := "<SELECTION>";
                        Stack_Frame : Integer     := 0);
```

Description

Enables or disables the recording of information about events executed in the specified task at a specified part of the program.

By default, this procedure causes history to be accumulated for all tasks for all events that occur in the selected location.

The Event parameter specifies the type of information to be recorded; the For_Task parameter specifies the task to be monitored; and the At_Location parameter specifies the subprogram or statement to be monitored.

The Debugger can keep a history of execution for each task. The history is the set of most recently executed statements, declarations, calls, rendezvous, and/or exceptions. The set is limited to approximately 1,000 entries.

Execution is slowed significantly when history taking is enabled.

History information, like traces, can include messages about statements, calls, rendezvous, and exception raising. Trace messages are displayed each time an event occurs. Unlike traces, history messages are saved in a circular buffer in the Debugger. From the buffer, selected sets of messages can be displayed, such as messages from only a specific task or for some range of messages. See the Trace procedure for more information on tracing.

The example, below, shows the output form for histories.

Parameters

```
On : Boolean := True;
```

Specifies whether to enable (turn on) or disable (turn off) the taking of the history for the specified task. The default is to enable taking the history.

```
Event : Trace_Event := Debug.All_Events;
```

Specifies the type of information to be recorded. By default, all events are recorded.

```
For_Task : Task_Name := "";
```

Specifies the task of which to take the history. The default is the task specified by the control context or, if the control context is not set to a specific task, all tasks.

```
At_Location : Path_Name := "<SELECTION>";
```

Specifies that the history that is gathered be restricted to the specified location. The null string ("") means all locations in the program. If the null string is not specified, the At_Location parameter must refer to a subprogram or statement.

The At_Location parameter is the primary means for specifying the location for which history should be recorded. The Stack_Frame parameter (see below) provides a convenient means for specifying a frame (subprogram) for which to record history. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Take_History command is bound to a key. If the Stack_Frame parameter is nonzero and the At_Location parameter specifies a special name (such as "<SELECTION>"), the At_Location parameter is ignored and history recording will be active in the subprogram specified by Stack_Frame. If the Stack_Frame parameter is nonzero and the At_Location parameter specifies a relative pathname, the actual pathname used to specify the location of the restriction is composed by appending the string "_*n*" to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the At_Location parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If the At_Location parameter is a special name that does not resolve to a location (for example, if the At_Location parameter is "<SELECTION>" and the cursor is not in the selection), history recording will occur everywhere.

```
Stack_Frame : Integer := 0;
```

Specifies the frame (subprogram) for which to collect history. The interpretation of this parameter is discussed in more detail in the description of the At_Location parameter above. By default, this parameter is ignored and history recording occurs at the location specified by the At_Location parameter.

Restrictions

A maximum of 20 requests for history taking can be set.

Example

This example shows that each line is displayed if all events are to be captured:

```
Call history-taking has been enabled for the specified location and task.

Exception history-taking has been enabled for the specified location
and task.

Statement history-taking has been enabled for the specified location
and task.
```

References

procedure History_Display

procedure Trace

type Trace_Event

type Task_Category

```
type Task_Category is (All_Tasks, Blocked, Held, Not_Running, Running,
                       Stopped);
```

Description

Defines the category or subset of tasks to be displayed by the Task_Display procedure.

The type allows the Task_Display procedure to display information about only a subset of all tasks in the current program.

Enumerations

`All_Tasks`

Specifies all tasks defined in the current program. The current program is defined as all tasks created within the job being debugged.

`Blocked`

Specifies all tasks currently some external event, typically a rendezvous or a delay, but not held or stopped in the Debugger.

`Held`

Specifies all tasks that are currently held in the Debugger (tasks are held by using the Hold procedure). These tasks would always be in the set of stopped tasks as well.

`Not_Running`

Specifies all tasks currently not executing for any reason. This set of tasks, stopped within the Debugger, includes those not running for Ada reasons, such as delays.

`Running`

Specifies all tasks currently ready to run. Tasks that are executing delay statements or waiting for a rendezvous are excluded, as are tasks stopped in the Debugger because of breakpoints and the like.

`Stopped`

Specifies all tasks that are currently stopped in the Debugger.

procedure Task_Display

```
procedure Task_Display (For_Task : Task_Name     := "";
                        Task_Set : Task_Category := Debug.All_Tasks);
```

Description

Displays information about the named task(s).

This procedure displays information about a set of tasks or a named task is displayed in the Debugger window.

If the For_Task parameter specifies a specific task, the Task_Set parameter is ignored. If For_Task specifies all tasks (with the string "all"), Task_Set may select some subset of all tasks.

The information displayed includes the Environment task number and string nickname for the task (if any), the Ada name of the task, the priority of the task, the current state of the task, and (if applicable) the location in which the task is stopped. The latter is included only if the task is stopped in the Debugger.

A string name for a task is set by a call to the Debug.Set_Task_Name procedure or to the Debug_Tools.Set_Task_Name procedure.

The state information listed for the task indicates whether the task is running, is being stopped, or is stopped. If the task is running, its execution state is listed. If it is stopped, the reason for the stop is listed. A sample display appears in the example below.

Terminated tasks are not listed.

The execution state messages displayed for running tasks are (the state name follows the description and is enclosed in parentheses):

- "": The task is ready to execute or is currently executing. Higher-priority tasks may prevent the task from running, but there is no condition associated with the task to keep it from running. (Unblocked)
- aborting task: The module is in the process of aborting a task it has declared. (Aborting_Module)
- activating child packages: The module is waiting for children packages to become elaborated. (Activating_Module)
- activating child tasks: The module is telling its child tasks to become activated. (Activating_Tasks)
- attempting entry call: Reserved for future use. (Attempting_Entry)

- being aborted: The task is in the process of making another task abnormal (see the *Reference Manual for the Ada Programming Language*, Section 9.10). (Blocking_In_Abort)
- delaying in wait service: The task is blocked in an Environment service and has a maximum wait time. (Delaying_In_Wait_Service)
- in wait service: The task is blocked in an Environment service. (In_Wait_Service)
- package completed: The task has reached its end and will terminate according to Ada rules when its parent decides it is time to terminate. (Terminable_At_End)
- state presently unknown: The task is processing a page fault. This transient state is unlikely to be seen. (In_Fs_Rendezvous)
- terminated: The task is terminated. (Terminated)
- waiting at accept for entry call: The task is blocked at an accept statement, waiting for an entry call. (Blocking_On_Accept)
- waiting at entry for accept: The task is blocked waiting for an entry call to be accepted. (Blocking_On_Entry)
- waiting at select for entry call: The task is blocked at a select statement, waiting for an entry call. (Blocking_On_Select)
- waiting at select-delay for entry call: The task is blocked in a select with a delay alternative waiting for an entry call. (Delaying_On_Select)
- waiting at select-terminate for entry call with dependents: The task is executing at a select, with a terminate alternative, and has dependent, nonterminable tasks. (Awaiting_Children_In_Select)
- waiting at select-terminate for entry call: The task is blocked in a select, with a terminate alternative, waiting for an entry call and can terminate. (Terminable_In_Select)
- waiting at timed entry for accept: The task is blocked at a timed entry call waiting for the called task to accept the call. (Delaying_On_Entry)
- waiting for child elaboration: The module is in the process of declaring a task or package. (Declaring_Module)
- waiting for children: The task is waiting for its children tasks to be terminated. (Awaiting_Children)
- waiting for delay: The task is blocked at a delay statement. (Delaying)
- waiting for parent elaboration: The module is waiting for a package or task it has declared to become activated. (Awaiting_Activation)
- waiting for task activation: The module has told its child tasks to become activated and is waiting for confirmation that each has been activated. (Awaiting_Task_Activation)

Parameters

```
For_Task : Task_Name := "";
```

Specifies a specific task about which to display information. The default is the control context task or, if the control context is not set, all tasks (or a possible subset selected by the Task_Set parameter; see below).

The string "all" can be used to specify all tasks (or a possible subset selected by the Task_Set parameter; see below).

```
Task_Set : Task_Category := Debug.All_Tasks;
```

Specifies a subset of the tasks about which to display information. The default parameter is all tasks. This parameter is ignored if the For_Task parameter specifies a specific task.

Errors

The named task does not exist.

Example

The following example illustrates some of the kinds of information that the Task_Display command produces:

```
Job:   212, Root task: #2DCD4
ROOT_TASK, #2DCD4 (Root task): Running, waiting for children. [Pri = 1]
QUEUE, #2FØD4 (.PRODUCER_CONSUMER.QUEUE): Stop at .PRODUCER_CONSUMER.
    QUEUE.7S. [Pri = 1]
PRODUCER, #2F4D4 (.PRODUCER_CONSUMER.PRODUCER): Stop at
    .PRODUCER_CONSUMER.PRODUCER.4S. [Pri = 1]
CONSUMER, #2F8D4 (.PRODUCER_CONSUMER.CONSUMER): Stop at
    .PRODUCER_CONSUMER.PRODUCER.4S. [Pri = 1]
```

The first line of the above display from the Debugger window indicates the job number of the program being debugged and the root task number of the job. The root task is the task that was created when the command was executed with debugging enabled.

Each entry (four are shown in the above display) begins with the task name defined in the program, if any, then the Environment task number, followed by the Ada name of the task in parentheses. Here, task #2F0D4 is declared as task Queue within program unit Producer_Consumer. Task #2DCD4 is the root task and has no Ada task declaration.

Task Queue is stopped at statement 7. Task #2DCD4 is currently not stopped, but it is not ready to run because it is waiting for one or more child tasks to be terminated.

subtype Task_Name

```
subtype Task_Name is String;
```

Description

Defines a hexadecimal number or a string representation of the name of a task in the program that is being debugged.

Many operations require a specific task to be named. This type defines the way in which those tasks are named. The names can take either of two forms: a hexadecimal number or a user-defined string.

Each task in the program is assigned a number by the Rational Environment. This number cannot be predicted. When a task stops in the Debugger, its number is reported (along with other information about the task). The Task_Display procedure lists all tasks in the job being debugged along with the number and Ada name of the task.

Tasks can also be assigned a string nickname. This assignment is made by the Set_Task_Name procedure in this package. This assignment is also made by having the task that is to have the name call the Debug_Tools.Set_Task_Name procedure. String names assigned to be task names must be legal Ada identifiers. In particular, these names must not contain periods.

Restrictions

The task name must be prefixed with the percent symbol (%). This character identifies the name as a task name.

An exception is made if the numeric form of the task name is used and the leading digit of the task number is a numeral (0 through 9). No other interpretation of the name is possible in this case, so the leading percent symbol (%) is optional.

References

procedure Break

procedure Execute

procedure Hold

procedure Run

procedure Set_Task_Name

procedure Stack

procedure Stop

procedure Task_Display

procedure Trace

procedure Xecute

procedure Debug_Tools.Set_Task_Name

---

procedure Trace

```
procedure Trace (On          : Boolean     := True;
                 Event       : Trace_Event := Debug.All_Events;
                 In_Task     : Task_Name   := "";
                 At_Location : Path_Name   := "<SELECTION>";
                 Stack_Frame : Integer     := Ø);
```

Description

Enables or disables the tracing of specified events in the named task.

By default, tracing will begin for all events in all tasks when the events occur at the selected location.

A trace displays information about the execution of the task in the Debugger window. The procedure defines and enables or disables these traces. Trace output can also be directed to a file using the Trace_To_File procedure.

Trace information, like histories, can include messages about statements, calls, rendezvous, and exception raising. Trace messages are displayed each time an event occurs. Unlike traces, history messages are saved in a circular buffer in the Debugger. From the buffer, selected sets of messages can be displayed, such as messages from only a specific task or for some range of messages. See the Take_History and History_Display procedures for more information on histories.

See the example below for a discussion of the output form.

Trace messages are displayed as follows:

- Statement trace messages are displayed before the statement is executed.
- Call trace messages are displayed before the first and after the last declaration or statement of the subprogram is executed.
- Rendezvous trace messages are displayed before the first statement of the rendezvous is executed.
- Exception trace messages are displayed immediately after the exception is raised but before any stack frames are popped and before the handler code is executed.

Parameters

```
On : Boolean := True;
```

Specifies whether to enable or disable the trace. The default is to enable the trace.

```
Event : Trace_Event := Debug.All_Events;
```

Specifies the class of execution events to be traced. The default is to trace all events.

```
In_Task : Task_Name := "";
```

Specifies the task to be traced. The default is the task specified by the current control context. If the control context is not set to a specific task, the default is all tasks.

```
At_Location : Path_Name := "<SELECTION>";
```

Specifies a restriction on the location in which tracing should be active. This parameter can specify a subprogram or a statement. Multiple Trace procedures can be executed to enable tracing in several subprograms. The null string ("") specifies that tracing should be enabled everywhere in the program being debugged. By default, tracing is performed only at the selected location.

The At_Location parameter is the primary means for specifying the location at which tracing should be performed. The Stack_Frame parameter (see below) provides a convenient means for specifying a frame in which to perform tracing. Typically, the value of the Stack_Frame parameter is provided using argument prefix keys when the Trace command is bound to a key. If the Stack_Frame parameter is nonzero and the At_Location parameter specifies a special name (such as "<SELECTION>"), the At_Location parameter is ignored and tracing is active in the subprogram specified by Stack_Frame. If the Stack_Frame parameter is nonzero and the At_Location parameter specifies a relative pathname, the actual pathname used to specify the location of the restriction is composed by appending the string "_*n*" to the value of the Path_Name parameter, where *n* is the value of the Stack_Frame parameter. If the At_Location parameter specifies an absolute pathname, the Stack_Frame parameter is ignored.

If the At_Location parameter is a special name that does not resolve to a location (for example, if the At_Location parameter is "<SELECTION>" and the cursor is not in the selection), tracing will occur everywhere.

```
Stack_Frame : Integer := 0;
```

Specifies the frame (subprogram) in which to perform tracing. The interpretation of this parameter is discussed in more detail in the description of the At_Location parameter above. By default, this parameter is ignored and tracing occurs at the location specified by the At_Location parameter.

Restrictions

A maximum of 20 tracing requests can be set.

Tracing substantially slows execution because messages must be displayed for each trace event. Thus, it is a good idea to use tracing only in regions of the program where information is required. Tracing can be done in part by stopping the program at the beginning of the region to be traced, enabling tracing, and then stepping some moderate number of statements. When the desired information is obtained, execution can be stopped and tracing disabled.

Tracing can be disabled only for a location in which it was enabled. The disabling of tracing at a specific location cannot be used to achieve the effect of tracing everywhere but in a specific location.

Example

If the commands:

```
Trace (True, Statement, "%432E");
Trace (True, Call, "%542E");
Trace (True, Exception_Raised);
```

are issued, the sample trace messages would be:

```
Statement trace at !USERS.DRK.TEST_JOB.4s     [Task : #432E].
Statement trace at !USERS.DRK.TEST_JOB.5s     [Task : #432E].
Call trace at !USERS.DRK.READY  [Task : #542E].
Call trace at !USERS.DRK.READY.SET     [Task : #542E].
Call trace at !USERS.DRK.READY.GO  [Task : #542E].
Call trace at !USERS.DRK.READY.GO.START     [Task : #542E].
Call trace at !USERS.DRK.WAIT  [Task : #542E].
Call trace at !USERS.DRK.POSTMORTEM     [Task : #542E].
Exception trace at !USERS.DRK.INTERCEPT.7s     [Task : #542E].
Exception was !USERS.DRK.OUT_OF_RANGE.
```

Each trace message identifies the type of trace, the current location of the executing task, and the task name. Exception messages also name the exception raised.

References

procedure History_Display

procedure Take_History

type Trace_Event

procedure Trace_To_File

---

type Trace_Event

```
type Trace_Event is (All_Events, Call, Exception_Raised,
                     Propagate_Exception, Rendezvous, Statement);
```

Description

Defines the class of events that are to be traced by the Trace and Take_History procedures.

Enumerations

`All_Events`

Specifies all events, including statements, exceptions, rendezvous, and subprogram calls, to be traced.

`Call`

Specifies subprogram calls and returns to be traced.

`Exception_Raised`

Specifies points where exceptions are raised to be traced.

`Propagate_Exception`

Specifies subprograms where exceptions are propagated to be traced.

`Rendezvous`

Specifies begin and end rendezvous to be traced.

`Statement`

Specifies all statements to be traced.

procedure Trace_To_File

```
procedure Trace_To_File (File_Name : String := ">> FILE NAME <<");
```

Description

Sends trace output to the file specified by File_Name parameter.

This procedure causes any existing file being used for tracing output to be closed.

If the File_Name parameter is null, subsequent output goes to the Debugger window. Thus, the command `Trace_To_File ("")` restores tracing output to the Debugger window and closes the previously selected output file.

Parameters

`File_Name : String := ">> FILE NAME <<";`

Specifies the file to which to send tracing output. The default parameter placeholder must be replaced with a legal file name or "" or an error message will be generated.

If File_Name is null, subsequent output goes to the Debugger window. Thus, the command `Trace_To_File ("")` restores tracing output to the Debugger window and closes the previously selected output file.

procedure Xecute

```
procedure Xecute (Name : Task_Name := "");
```

Description

Commences (or resumes) execution of the named task(s).

This command is functionally equivalent to the Execute procedure.

The named task starts executing from its current location—that is, from where it was stopped because of a breakpoint or a Stop or Hold procedure, an exception being trapped, or the end of a stepping request. If the task is already executing, the procedure has no effect.

If a specific task is named and that task is being held (by means of the Hold procedure), then the hold condition is removed.

If the Name parameter is "all", any task that is stopped for any reason in the Debugger is allowed to continue unless the task has a hold on it. Tasks subject to hold conditions must be started individually by name, or the hold condition must be released with the Release procedure.

If the Freeze_Tasks flag is true and all tasks are stopped implicitly as a result of an individual task being stopped, the Xecute procedure commences execution of these implicitly stopped tasks.

Parameters

`Name : Task_Name := "";`

Specifies the task to be executed. The default is the task specified by the control context or all nonheld tasks if the control context is not explicitly set.

The reserved word "all" can be used to specify that all nonheld tasks are to executed.

See the Hold procedure for more information on the held state.

Errors

A `No tasks are stopped` message occurs when no tasks are stopped in the Debugger or when the only tasks stopped are subject to hold conditions.

References

procedure Break

procedure Execute

procedure Hold

procedure Release

procedure Stop

end Debug;

package Debug_Tools

Package Debug_Tools provides a programmatic interface to the Debugger. These subprograms can be used in a program to pass information back to the Debugger. They can also be used to determine information about the task calling the subprograms, including the most recent exception raised, the current program counter location, the Debugger task name synonym, and so on. The subprograms also provide a means for creating special display procedures for the Debugger to use to display the value variables of specific types (for example, when using the Debug.Put command), where users do not want the Debugger to use the structural type information for displaying these values.

The preceding section describes package !Commands.Debug, which contains the interactive commands for Debugger operations.

function Ada_Location

```
function Ada_Location (Frame_Number   : Natural := 0;
                       Fully_Qualify  : Boolean := True;
                       Machine_Info   : Boolean := False) return String;
```

Description

Returns a string representing the Ada name of the source location in the calling subprogram or a caller of the calling subprogram.

The null string is returned if the frame specified by the Frame_Number parameter does not exist.

Parameters

`Frame_Number : Natural := 0;`

Specifies, when Frame_Number = 0, that the source location in the caller of Ada_Location be returned. Frame_Number = 1 specifies that the source location in the caller of the caller of Ada_Location be returned, and so on.

`Fully_Qualify : Boolean := True;`

Specifies, when true, that the name returned be fully qualified. Otherwise, only the simple name of the subprogram is returned.

`Machine_Info : Boolean := False;`

Specifies, when true, that machine information, such as the program counter, be returned as part of the string.

`return String;`

Returns the Ada name of the location. The value of the string may be relatively long, especially if Fully_Qualify and Machine_Info are both true. The string contains blanks but no embedded control characters.

Restrictions

Frames for which no name is available cause the return of a nonnull string that states that no name is available.

If there is no Debugger for the session (that is, if no jobs have been debugged since the user logged in), this function returns the string "Unknown - Debugger not started".

Example

If a task issues the call:

```
Io.Put_Line (Debug_Tools.Ada_Location);
```

the output displayed in the output window is:

```
.DEBUG_TOOLS_EXAMPLES.4s
```

procedure Debug_Off

```
procedure Debug_Off;
```

Description

Disables debugging in the calling task.

This command allows a program to turn off debugging so that the Debugger does not interfere with program behavior. When a program is being debugged, it can turn off debugging by calling the Debug_Off procedure; when the program desires to allow debugging again, it can call the Debug_On procedure. When debugging is off, you are not able to control execution using the Debugger (including catching exceptions and so on).

Restrictions

If this command is called from a job not currently being debugged, the call is ignored and has no effect.

References

procedure Debug_On

function Debugging

procedure Debug_On

```
procedure Debug_On;
```

Description

Enables debugging for the calling task.

Used only after the Debug_Off procedure has been called, the command allows a program to implement a region in which the Debugger will not interfere with program execution. Such regions can be created by calling the Debug_Off procedure at the beginning of the region to tell the Debugger not to interfere until the next call to Debug_On, and then calling the Debug_On procedure at the end of the region to resume Debugger control. During execution in this region, you are not able to control execution using the Debugger (including catching exceptions).

Any tasks created from a region in which debugging is not enabled also do not have debugging enabled. Once the Debug_On procedure has been called, tasks subsequently created will have debugging enabled.

Restrictions

If called from a job not currently being debugged, the call is ignored and has no effect.

References

procedure Debug_Off

function Debugging

function Debugging

```
function Debugging return Boolean;
```

Description

Returns true if the currently executing program is under the control of the Debugger; otherwise, the function returns false.

Parameters

`return Boolean;`
Returns true if the currently executing program is under the control of the Debugger; otherwise, the function returns false.

References

procedure Debug_Off

procedure Debug_On

EST, procedure Command.Debug

function Get_Exception_Name

```
function Get_Exception_Name (Fully_Qualify : Boolean := True;
                             Machine_Info  : Boolean := False)
                                                    return String;
```

Description

Returns the name of the most recently raised exception for the task that calls this function and, optionally, returns additional machine-related information about the exception.

This function must be called directly or indirectly from an exception handler. The null string is returned if no exception is currently active for the calling task.

If Machine_Info is true, the address in which the exception was raised and the exception's machine representation are displayed. These numbers can be given to the Debug.Address_To_Location and Debug.Exception_To_Name procedures to get their source representations (assuming that the Ada units still exist and the exception was not raised or declared in a Command window).

Parameters

`Fully_Qualify : Boolean := True;`

Specifies, when true, that the exception name returned be fully qualified.

`Machine_Info : Boolean := False;`

Specifies, when true, that machine information, such as the program counter in which the exception is raised, be included in the string.

`return String;`

Returns information about the exception. The string may be relatively long (that is, greater than 80 characters), especially if Fully_Qualify and Machine_Info are both true.

Example

If a task is in an exception handler because of a Program_Error exception and issues the call:

```
Io.Put_Line (Debug_Tools.Get_Exception_Name);
```

the output displayed in the output window is:

```
Program_Error
```

References

procedure Debug.Address_To_Location

procedure Debug.Exception_To_Name

---

function Get_Raise_Location

```
function Get_Raise_Location (Fully_Qualify : Boolean := True;
                             Machine_Info  : Boolean := False)
                                                    return String;
```

Description

Returns a representation of the location in the source from which the most recent exception for the task calling this function was raised and, optionally, returns additional machine-related information about the raise location.

This function must be called directly or indirectly from an exception handler. The null string is returned if no exception is currently active for the calling task or if problems are encountered getting the raise location information.

If Machine_Info is true, the address in which the exception was raised and the exception's machine representation are displayed. These numbers can be given to the Debug.Address_To_Location and Debug.Exception_To_Name procedures to get their source representations (assuming that the Ada units still exist and the exception was not raised or declared in a Command window).

Parameters

`Fully_Qualify : Boolean := True;`

Specifies, when true, that the raise location returned be fully qualified.

`Machine_Info : Boolean := False;`

Specifies, when true, that machine information, such as the program counter address in which the exception was raised, be included in the string.

`return String;`

Returns the location in the source from which the most recent exception for the calling task was raised. The string may be relatively long (that is, greater than 80 characters), especially if Fully_Qualify and Machine_Info are both true.

Restrictions

If there is no Debugger for the session (that is, if no jobs have been debugged since the user logged in), this function will return only low-level machine information.

Example

If a task in a program being debugged with a number 1CCD4 is in an exception handler because of a Program_Error exception and issues the call:

```
Io.Put_Line (Debug_Tools.Get_Raise_Location);
```

the output displayed in the output window is:

```
Task ROOT_TASK, #1CCD4:
    Exception Program_Error raised at: .DEBUG_TOOLS_EXAMPLES.5s.
```

References

procedure Debug.Address_To_Location

function Get_Task_Name

```
function Get_Task_Name return String;
```

Description

Returns any task name set by the Set_Task_Name procedure.

This function allows a task to interrogate its name. If the function is called by a task not in a job presently being debugged, the name passed to Set_Task_Name may or may not be returned.

Parameters

`return String;`
Returns whatever is passed to the Set_Task_Name procedure. The null string is returned if no name has been assigned.

Example

If a main program with no tasks is being debugged and issues the call:

```
Io.Put_Line (Debug_Tools.Get_Task_Name);
```

the output displayed in the output window is:

```
Root_Task
```

If that program had first set the name of its root task to "Main" by calling the Set_Task_Name procedure, the following output would have been displayed:

```
Main
```

Restrictions

If there is no Debugger for the session, this function always returns the null string ("").

function Get_Task_Name
package !Tools.Debug_Tools

References

procedure Set_Task_Name

procedure Debug.Set_Task_Name

---

procedure Message

```
procedure Message (Info : String);
```

Description

Causes a message to be displayed in the Debugger window.

A call to the Message procedure from any task being debugged causes a message as specified by the Info parameter to be displayed in the Debugger window.

If there is no Debugger for the session, the operation has no effect.

Parameters

`Info : String;`

Specifies the string that is displayed in the Debugger window. Multiple lines should be separated with Ascii.Lf characters.

Example

If a task with number 1CCD4 issues the call:

```
Debug_Tools.Message ("here is a message");
```

the message displayed in the Debugger window is:

```
From task: ROOT_TASK, #1CCD4: here is a message
```

RATIONAL

generic procedure Register

This generic procedure provides facilities that enable you to write *special display* routines for the Debugger. The Debugger uses the routines to display the value of variables and to perform other actions. These display routines can be created for any type defined in the Rational Environment or in your applications.

The display routines are created and then registered with the Debugger. Once a special display is registered for a type, the Debugger uses it (instead of the structural type information from the type declaration) when the Debug.Put command is executed on variables of that type. Registering a new special display for a type overrides any existing special displays for the type and the type's normal structural display. Special displays can be unregistered to enable the use of a type's normal structural display.

Special displays can be registered when the Debugger is started in the default Debugger_Initialization procedure. If they are registered in this procedure, they will remain in effect until explicitly unregistered, until a new special display is registered for the type, until the user logs off from the session, or until the Debugger is killed. Special displays can also be registered by the job being debugged. In this case, the special display remains in effect during the life of the job, until the special display is unregistered, or until a new special display is registered for the type.

Note that, by default, the Debugger has preregistered special displays for many of the important types defined in the specifications of the Rational Environment. These displays can be redefined if necessary. The types that have displays registered for them include:

!Implementation.Activity_Implementation.Activity_Handle

!Implementation.Activity_Implementation.Iterator

!Implementation.Dependency_Data_Base.Iterator

!Implementation.Dependency_Data_Base.Defid_Iterator

!Implementation.Diana.Attr_List

!Implementation.Diana.Attr_Name

!Implementation.Diana.Comment

!Implementation.Diana.Number_Rep

!Implementation.Diana.Sequence
!Implementation.Diana.Seq_Type
!Implementation.Diana.Symbol_Rep
!Implementation.Diana.Temp_Seq
!Implementation.Diana.Tree
!Implementation.Diana.Value
!Implementation.Directory.Ada.Unit
!Implementation.Directory.Class
!Implementation.Directory.Naming.Iterator
!Implementation.Directory.Object
!Implementation.Directory.Object_Set.Iterator
!Implementation.Directory.Object_Set.Set
!Implementation.Directory.Traversal.Associated_Object_Iterator
!Implementation.Directory.Traversal.Object_Iterator
!Implementation.Directory.Traversal.Subobject_Iterator
!Implementation.Directory.Traversal.Subunit_Iterator
!Implementation.Directory.Traversal.Version_Iterator
!Implementation.Directory.Version
!Implementation.Error_Messages.Annotation
!Implementation.Error_Messages.Errors
!Implementation.Links_Implementation.Iterator
!Implementation.Low_Level_Action.Id
!Implementation.Switch_Implementation.Iterator
!Implementation.Universal.Float
!Implementation.Universal.Int
!Implementation.Universal.Integer
!Implementation.Universal.Real
!Io.Device_Independent_Io.File_Type
!Io.Io.File_Type
!Io.Object_Set.Iterator
!Io.Object_Set.Set
!Io.Pipe.Handle
!Io.Polymorphic_Io.File_Position
!Io.Polymorphic_Io.Handle
!Io.Polymorphic_Sequential_Io.File_Type

!Io.Text_Io.File_Type
!Io.Window_Io.File_Type
!Lrm.Calendar.Time
!Tools.Bounded_String.Variable_String
!Tools.Directory_Tools.Object.Error_Code
!Tools.Directory_Tools.Object.Handle
!Tools.Directory_Tools.Object.Iterator
!Tools.Directory_Tools.Object.Message_List
!Tools.Directory_Tools.Object.Subclass
!Tools.Link_Tools.Dependent_Iterator
!Tools.Link_Tools.Link_Iterator
!Tools.Profile.Response_Profile
!Tools.Simple_Status.Condition
!Tools.Simple_Status.Condition_Name
!Tools.String_Table.Item
!Tools.String_Table.Iterator
!Tools.System_Utilities.Job_Iterator
!Tools.System_Utilities.Session_Iterator
!Tools.System_Utilities.Terminal_Iterator
!Tools.Tape_Tools.Logical_Device
!Tools.Unbounded_String.Variable_String

Several important restrictions that apply to special displays are described under "Restrictions" in the reference entry for the Image generic formal function.

The formal parameters to the generic are:

```
generic
    type T is limited private;
    with function Image (Value           : T;
                         Level           : Natural;
                         Prefix          : String;
                         Expand_Pointers : Boolean) return String;
procedure Register;
```

These parameters define the type for which a special display is to be created and the function that returns the images of values of that type. This function is used by the Debugger when performing puts on values of that type. Instantiating the generic provides the registration procedure that can be called from your application (or by your Debugger_Initialization procedure) to register the special display.

Example 1

Even though there may be a structural display available for a type, a special display may be more convenient to use. For example, data structures containing pointers are often difficult to visualize from the structural display. The following example shows how a special display can be constructed for lists. This special display outputs the elements in the list as a text string instead of the pointer chain connecting the list elements, as would typically be the case if a structural display were used.

```
with List_Generic;
package Integer_List is new List_Generic (Integer);
-- Example uses list generic from !Tools.


with Integer_List;
package List_Display is

    -- This package implements the special display for values
    -- of type Integer_List.List.

    procedure Register;
    -- Causes the special display to be registered with the
    -- Debugger so that it will be used by the Debugger in
    -- subsequent Debug.Put commands instead of the normal
    -- structural display.

end List_Display;


with Debug_Tools;
with String_Utilities;
package body List_Display is

    function Element_Images (Value : Integer_List.List) return String is
    begin
        if Integer_List.Is_Empty (Value) then
            return "";
        else
            return String_Utilities.Number_To_String
                       (Integer_List.First (Value)) & " " &
                   Element_Images (Integer_List.Rest (Value));
        end if;
    end Element_Images;
    -- Recursive function that builds a string containing the
    -- the images of the items in a list ordered from left to
    -- right.

    function Image (Value : Integer_List.List;
                    Level : Natural;
                    Prefix : String;
                    Expand_Pointers : Boolean) return String is
    begin
        if Integer_List.Is_Empty (Value) then
            return "the list is empty";
        else
            return "( " & Element_Images (Value) & ")";
        end if;
    end Image;
    -- The Image function used in this special display displays
```

```
        -- lists with their elements ordered from left to right
        -- enclosed in parentheses.  E.g., "( 1 2 3 )".

        procedure Register_Special_Display is
           new Debug_Tools.Register (Integer_List.List, Image);

        procedure Register is
        begin
            Register_Special_Display;
        end Register;

    begin
        Register;  -- Registers the special display upon elaboration.
    end List_Display;


    with Text_Io;
    with Integer_List;
    with List_Display;
    procedure List_Test is

        -- This is a main program used to test the special display
        -- for Integer_List.List.

        L : Integer_List.List := Integer_List.Nil;

    begin

        L := Integer_List.Make (3, L);
        L := Integer_List.Make (2, L);
        L := Integer_List.Make (1, L);

        null;  -- A location on which to stop the Debugger.

    end List_Test;
```

Assume that the List_Test procedure is run with the Debugger and stepped to the null statement. If the Debug.Put command is executed at this point to display the value of L, the Debugger responds as follows in the Debugger window:

```
Put ("%ROOT_TASK._1.L");
(( 1 2 3 ))
```

Example 2

Sometimes it is helpful in testing an application program to be able to initiate the execution of subprograms as part of the job being debugged. For example, when debugging an application using the Debugger, it might be helpful to be able to reset some of the state of the application, display status information, or dump/restore state, all by executing Debugger commands at certain times. The special display facility can be used to do this.

The strategy consists of defining types and objects of these types for each of the operations that you would like to trigger from the Debugger. Special displays are then created and registered for each of these types. These special displays can be designed to perform the actions desired. Finally, to invoke the special display and

to perform the action while debugging, simply execute the Debug.Put command on the object corresponding to the action you want to perform. The Debugger will execute the associated special display routine that will perform the action.

Here is a simple example of a package that implements such a strategy. Note that the package would be *with*ed into the library unit closure of the application. In this example, the only effect of the operations is to write a message into a log file. These actions typically would manipulate various pieces of the state of the application.

```
package Debugger_Operations is

    -- This package implements various testing operations using
    -- the Debugger special display mechanism.  To perform the
    -- desired action, perform a Debug.Put command on the object
    -- of the appropriate type.

    type Reset_Type is new Boolean;
    type Display_Status_Type is new Boolean;
    type Dump_State_Type is new Boolean;
    type Restore_State_Type is new Boolean;


    -- Performing a Debug.Put on the following objects causes the
    -- associated action to be performed by the special display
    -- for their types.

    Reset          : Reset_Type := True;

    Display_Status : Display_Status_Type := True;

    Dump_State     : Dump_State_Type := True;

    Restore_State  : Restore_State_Type := True;

end Debugger_Operations;


with Text_Io;
with Debug_Tools;
package body Debugger_Operations is

    File : Text_Io.File_Type;

    Resetting : constant String := "... Resetting";
    Displaying_Status : constant String := "... Displaying status";
    Dumping_State : constant String := "... Dumping state";
    Restoring_State : constant String := "... Restoring state";

    function Reset_Implementation (Value : Reset_Type;
                                   Level : Natural;
                                   Prefix : String;
                                   Expand_Pointers : Boolean)
                                  return String is
    begin
        Text_Io.Put_Line (File, Resetting);
        return Resetting;
    end Reset_Implementation;
```

```
    function Display_Status_Implementation
                (Value : Display_Status_Type;
                 Level : Natural;
                 Prefix : String;
                 Expand_Pointers : Boolean) return String is
    begin
        Text_Io.Put_Line (File, Displaying_Status);
        return Displaying_Status;
    end Display_Status_Implementation;

    function Dump_State_Implementation
                (Value : Dump_State_Type;
                 Level : Natural;
                 Prefix : String;
                 Expand_Pointers : Boolean) return String is
    begin
        Text_Io.Put_Line (File, Dumping_State);
        Text_Io.Close (File);
        return Dumping_State;
    end Dump_State_Implementation;

    function Restore_State_Implementation
                (Value : Restore_State_Type;
                 Level : Natural;
                 Prefix : String;
                 Expand_Pointers : Boolean) return String is
    begin
        Text_Io.Put_Line (File, Restoring_State);
        return Restoring_State;
    end Restore_State_Implementation;

    procedure Special_Display_Reset is
       new Debug_Tools.Register (Reset_Type, Reset_Implementation);

    procedure Special_Display_Display_Status is
       new Debug_Tools.Register (Display_Status_Type,
                                 Display_Status_Implementation);

    procedure Special_Display_Dump_State is
       new Debug_Tools.Register (Dump_State_Type,
                                 Dump_State_Implementation);

    procedure Special_Display_Restore_State is
       new Debug_Tools.Register (Restore_State_Type,
                                 Restore_State_Implementation);

begin

    Special_Display_Reset;
    Special_Display_Display_Status;
    Special_Display_Dump_State;
    Special_Display_Restore_State;

    Text_Io.Create (File, Text_Io.Out_File, "$log_file");

end Debugger_Operations;
```

Example 3

For debugging applications containing types with complex structures, it is often desirable to obtain different displays of the images of these types. For example, you may want to obtain more or less detail, or you may want to look at different classes of information at different times during the debugging session. There are two approaches to using special displays to accomplish this.

One approach is to use variables in the special display packages to indicate the level of detail or kind of information desired. The Image functions implementing the special display for a type can read these variables and, depending on their values, perform different actions. The values of these variables can be manipulated by the Debug.Modify command or by the special displays, as described in Example 2 above.

The other approach is to create multiple special displays and register and unregister them dynamically at various points in your application. Note that this registration/unregistration can be accomplished explicitly in the application or by the special displays, as described in Example 2 above.

generic formal function Image

```
with function Image (Value           : T;
                     Level           : Natural;
                     Prefix          : String;
                     Expand_Pointers : Boolean) return String;
```

Description

Returns the string that is the image of the value of the Value parameter for the special display for T.

Parameters

`Value : T;`

Specifies the value for which the image is to be computed.

`Level : Natural;`

Specifies the number of levels of detail to be displayed (determined dynamically by the nesting level of Value in the structure being displayed by the Debugger and the value of the Display_Level flag). If Level = 0, the entire value should be elided.

`Prefix : String;`

Specifies the prefix string to be appended to the beginning of any lines displayed after an Ascii.Lf line terminator character.

`Expand_Pointers : Boolean;`

Specifies, if true, that internal pointers should be expanded in the image (determined dynamically by the nesting level of Value in the structure being displayed by the Debugger and the value of the Display_Numeric flag).

`return String;`

Returns the image of the value of the Value parameter.

The string may contain Ascii.Lf characters to indicate that the image spans multiple lines. If Ascii.Lf characters are returned, the following line should be prefixed with the string "Prefix".

Restrictions

The Image function cannot call the Debugger (specifically, it cannot call any of the subprograms in packages Debug or Debug_Tools), and the code in the Image function cannot be debugged using the Debugger when implicitly called as a result of performing the Debug.Put command.

The image function currently is executed on the Debugger job thread and not the job for the application. Consequently, the Image function is treated as part of the Debugger job from a scheduling perspective. If the Image function is resource-intensive (or goes into an infinite loop—see "Errors," below), this may mean that the system becomes heavily loaded during execution of the Image function.

Another implication of execution on the Debugger job thread is that the Image function cannot depend on job-dependent state in the Rational Environment. For example, the Standard_Output and Current_Output files from package !Io.Text_Io are job specific. If an Image function tries to perform a call to the Text_Io.Put_Line procedure using the value of Current_Output for the file, the output will not go to Current_Output for your application; instead, it will go to Current_Output for the Debugger. Other examples of job-dependent state are current context, current profile, and current activity.

Errors

If the Image function goes into an infinite loop, the Debug.Put command that caused the call to the Image function will never complete. If this situation occurs, perform the Debug.Kill command to terminate debugging of the job and start over after removing the infinite loop from the special display.

References

type Debug.Numeric, enumeration Display_Level

type Debug.Numeric, enumeration Pointer_Level

---

procedure Register

```
procedure Register;
```

Description

Registers a special display for a type with the Debugger.

Registering a special display causes the Debugger to use your own display instead of the default the Debugger uses when values with the Debug.Put command. Your display is the image function that is supplied as the value for the Image generic formal function in an instantiation of the Register procedure. This function is used by the Debugger to compute the images of values of the type provided as the T generic formal type in the instantiation of the Register procedure. See the introduction to the Register generic procedure for more information on special displays.

If another special display is currently registered for the type, it is unregistered and the new function is substituted.

Note that, by default, the Debugger has preregistered special displays for many of the important types defined in the specifications of the Rational Environment. These displays can be redefined if necessary. For a list of these types, see the introduction to the reference entry for the Register generic procedure.

The Debugger command `Debug.Show (Debug.Special_Types)` can be used to get a list of the special displays that have been registered by the user for the job currently being debugged. Note that this list will not include the Environment types automatically registered by the Debugger. See the introduction to the Register generic procedure for a list of the preregistered Environment types.

Restrictions

Many important restrictions apply to the Image function being registered. See the "Restrictions" section of the reference entry for the Image generic formal function for these restrictions.

The source code for instantiations of the Register procedure must be on the system on which the program being debugged is executing. This is an important restriction to consider when using software that has been code-archived. The code-archived software that is distributed can include the image functions needed for registration. However, the actual source code for the registration must exist on the system receiving the code-archived software.

procedure Register
package !Tools.Debug_Tools

References

generic formal function Image

procedure Debug.Show

type Debug.State_Type, enumeration Special_Types

---

generic formal type T

```
type T is limited private;
```

Description

Specifies the type for which the special display is being defined.

end Register;

procedure Set_Task_Name

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

Description

Assigns a string *nickname* for the named task.

The name can be used by the user in various Debugger commands as a synonym for the task. If the command is called by a task in a job not presently being debugged, the call has no effect. The nickname can also be set by the Debug.Set_Task_Name procedure.

To use such a nickname in a command, the name must be preceded by a percent symbol (%). The string passed to this procedure, however, should not have a leading percent symbol.

It is good practice to call the Set_Task_Name procedure in important (if not all) tasks to identify them easily during debugging. The call is especially important when multiple instances of the same task are created. It is some extra work to give each a unique name, but the effort often greatly simplifies the task of understanding what is going on during debugging.

If the name passed to the Set_Task_Name procedure has already been used by a different task, the name is removed from that old task and assigned to the present caller to the Set_Task_Name procedure. Thus, only one task has a given name at a given time.

The name Root_Task is automatically assigned to the root task (the command task) in a job. To avoid confusion, it is best not to reassign this name.

Parameters

```
Name : String;
```

Specifies the name to be assigned to the task. It is best to limit the name to 40 characters, because Debugger displays that include the task name are less readable if excessively long names are used.

The string must be a legal Ada identifier. If the string contains illegal characters, it will be truncated to the longest leftmost substring that is a legal identifier.

Restrictions

This procedure has no effect if the job is not being debugged.

References

function Get_Task_Name

subtype Debug.Path_Name

procedure Debug.Set_Task_Name

procedure Debug.Task_Display

---

generic procedure Un_Register

This generic procedure causes a special display registered for a type to no longer be used by the Debugger for displaying objects of the type. When a special display is unregistered, the Debugger will use the structural information in the type declaration to display objects of these types. For more information on special displays, see the Register generic procedure.

The formal parameters to the generic are:

```
generic
    type T is limited private;
procedure Un_Register;
```

The parameter defines the type for which a special display is no longer to be used.

generic formal type T

```
type T is limited private;
```

Description

Specifies the type for which the special display is to be removed.

procedure Un_Register

```
procedure Un_Register;
```

Description

Causes the Debugger to stop using the Image procedure last registered for computing the images of values of the T generic formal type in calls to the Debug.Put command for values of T type.

After the call to Un_Register, the default display will be based on the structural information defining T.

If no special displays are registered for the type, this procedure has no effect.

References

generic formal function Image

generic procedure Register

end Un_Register;

procedure User_Break

```
procedure User_Break (Info : String);
```

Description

Causes the calling task to stop as though a breakpoint were reached.

The string specified by the Info parameter is displayed in the Debugger window along with a message that the task has stopped for a user break.

Parameters

```
Info : String;
```

Specifies the string that is displayed in the Debugger window. Multiple lines should be separated by Ascii.Lf characters.

Example

In the following example, the break message appears as:

```
From task: ROOT_TASK, #1CCD4: Serious error encountered
User break: .DAEMON.CRASH.3s  [Task : ROOT_TASK, #1CCD4].
```

which identifies the location and task name of the call to the User_Break procedure. The string passed as the Info parameter was "Serious error encountered".

Restrictions

If called by a task in a job that is not presently being debugged, this procedure has no effect.

end Debug_Tools;

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

C

D

E

F

G

H

I

J

K

L

M

O

Q

R

S

T

U

V

W

X

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