Rational Environment Reference Manual

System Management Utilities (SMU)

Document Control Number: 8001A-30

Rev. 0.0, November 1985
Rev. 1.0, April 1986
Rev. 2.0, July 1986
Rev. 3.0, July 1987 (Delta)

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 System Management Utilities (SMU) book of the *Rational Environment Reference Manual* contains reference information describing commands and tools provided by the Rational Environment™ that are useful primarily for system managers. This reference information is intended for users who are familiar with the Environment and Ada® programming. Note that the user-oriented commands in packages Operator and Queue are also documented in the Session and Job Management (SJM) book of the *Rational Environment Reference Manual.*

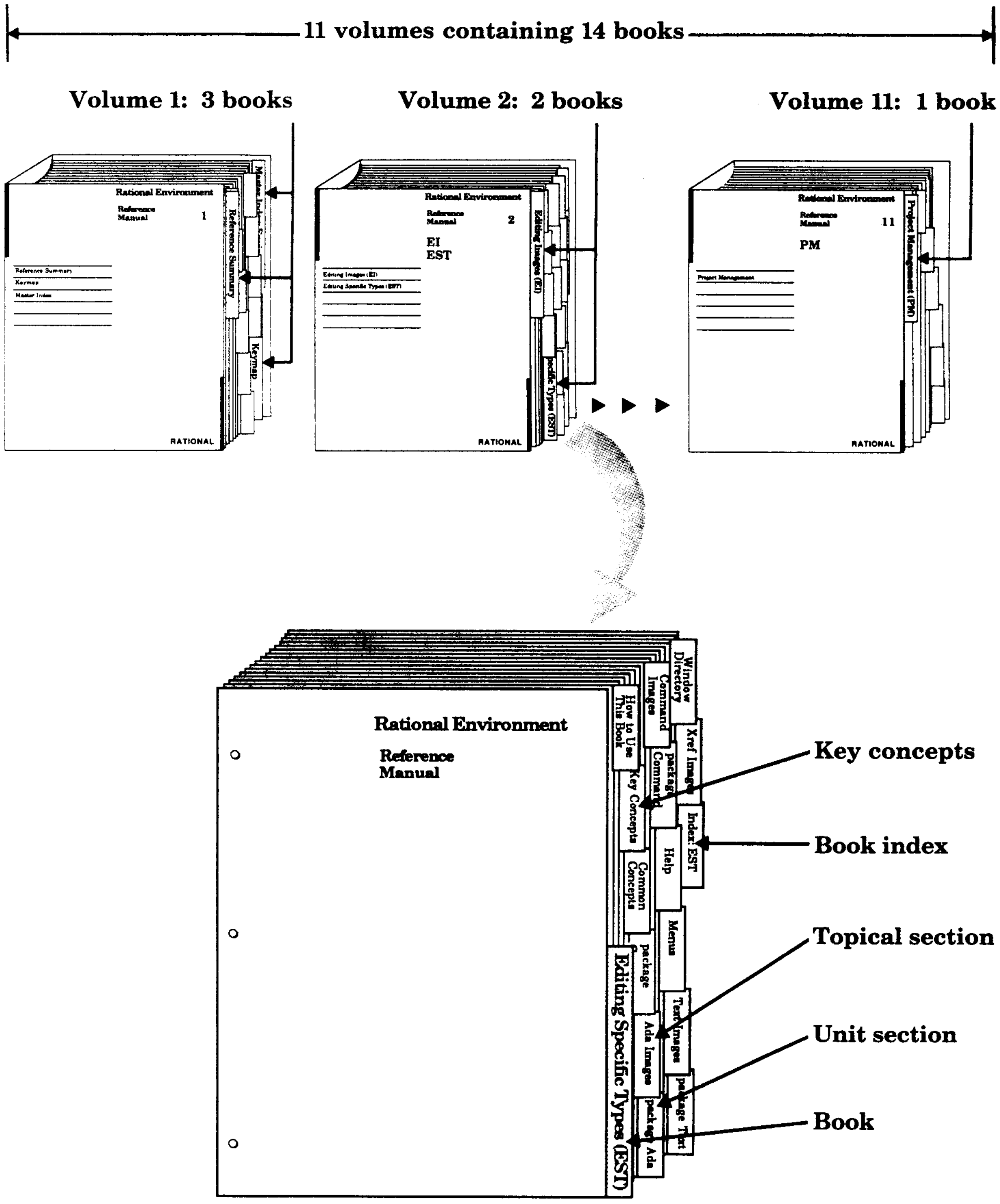
Organization of the Reference Manual

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

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

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

Organization of the *Rational Environment Reference Manual*



A sample book

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

Volume 1

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

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

Volumes 2–11

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

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

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

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

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

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

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

Suggestions for Finding Information

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

Learning about Environment Facilities

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

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

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

Finding Information on a Specific Item

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

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

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

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

Using the Index

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

Viewing Specifications On-Line

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

Using On-Line Help

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

Cross-Reference Conventions

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

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

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

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

Feedback to Rational: Reader's Comments Form

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

Key Concepts

System Management Utilities (SMU) documents the following packages, which are useful to system managers:

- Daemon: Utilities for maintaining system efficiency.
- Message: Utilities for sending messages.
- Operator: Commands for creating and deleting user accounts and groups.
- Queue: Commands for setting up and querying the print spooler.
- Scheduler: Utilities for fine-tuning system response.
- System_Backup: Commands for performing system backups.
- System_Utilities: Commands for gaining access to system characteristics that are set by packages !Commands.Job (documented in SJM), Operator, and Terminal.
- Tape: Commands for performing tape operations.
- Terminal: Commands for configuring, enabling, and disOpabling terminal ports.

Naming Objects

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

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

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

Special Names

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

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

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

Special Values

Many operations in the Environment have a Response parameter that specifies how the command should respond to errors.

Error Reactions

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

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

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

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

Parameter Placeholders

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

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

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

Wildcards

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

The Wildcard

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

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

The Wildcard @

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

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

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

The Wildcard ?

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

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

The Wildcard ??

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

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

Substitution Characters

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

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

The Substitution Character

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

The Substitution Character @

The *at* sign (@) is replaced by the portion of the current segment that is matched by a wildcard in the source name. If there is more than one wildcard in the segment, a separate @ is needed in the target to match each one. If the current segment has no wildcards, the next character that is followed by any of the special (not wildcard) characters covered in this section is not eligible as the source of the substitution. (For the purpose of this matching, @, #, ?, and ?? are considered to be wildcards.)

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

The Substitution Character ?

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

The Substitution Character ??

The double question mark (??) is replaced by full segments, including worlds. In the example in the previous paragraph, the target string !Users.John?? would copy everything in all subworlds.

Special Characters in Names

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

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

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

The Special Character !

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

The Special Character ^

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

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

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

The Special Character $

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

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

The Special Character $$

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

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

The Special Character %

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

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

The Special Character _

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

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

The Special Character .

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

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

**The Special Character **

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

More information about searchlists can be found in Session and Job Management (SJM).

The Special Character `

The grave (`) is used to evaluate names using the current context and the set of links associated with the current context. The grave evaluates the name as if it were the name of an Ada unit in a *with* clause of a unit in the library that contains the current context. For example, the name `Moe resolves to an Ada unit called Moe in the containing library. Moe could be a link to some other library.

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

More information about links can be found in Library Management (LM).

The Special Characters []

Brackets ([]) define a set notation. Sets are created by enclosing a series of name components, separated by commas, in brackets. For example, the name [Larry, Curly, Moe] represents only those three objects in the current context.

The semicolon character can also be used to separate name components. Commas and semicolons cannot be mixed. If semicolons are used, each name component in the set must resolve to at least one object. For example, Foo?['C(Lib), 'Spec] matches any component of Foo that is either a library or an Ada spec. Foo[A;B] must match A and B in Foo.

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

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

The Special Characters { }

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

The Options Parameter

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

Syntax Rules

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

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

"AFTER:=12/25/86"

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

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

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

Options taking string values that contain a comma or semicolon character must have the string enclosed in parentheses. For example:

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

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

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

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

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

Boolean Options: A Special Case

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

```
"REPLACE"

"REPLACE=>TRUE"

"REPLACE:=TRUE"

"REPLACE=TRUE"
```

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

```
"~REPLACE"

"REPLACE=>FALSE"

"REPLACE:=FALSE"

"REPLACE=FALSE"
```

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

```
"REPLACE PROMOTE"
```

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

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

Literals in Options: A Special Case

For literals of the form *literal* = *value*, the literal and value delimiter are optional.

RATIONAL

package Daemon

An integral part of the Rational Environment is a program called the system *daemon*, which periodically runs a set of special jobs that serve as system custodians. These special jobs, called *clients*, maintain system efficiency by managing the system's internal data structures, disposing of obsolete data, and reclaiming used storage space.

Using commands from package Daemon, you can schedule how often the system daemon runs its clients. You can also run clients independently of the schedule, prevent a scheduled client from running, and display information about each client. Using operations from package !Tools.Disk_Daemon, you can control more specifically the conditions under which the disk client runs.

Execution of some of the operations in this package requires that the executing job have operator capability. This is noted in the reference entry if the requirement applies.

The Daemon and Its Clients

When users create and modify objects such as Ada programs, text files, and directories, or when objects such as user accounts and sessions are modified, changes are made to data structures within the Environment. Each such data structure is managed by an *object manager* that normalizes it (removes obsolete data) and compacts it (reclaims the storage space that was used by the obsolete data).

The object managers include:

Actions	Ada	Archived_Code
Code_Segment	Configuration	DDB
Directory	File	Group
Link	Null_Device	Pipe
Session	Tape	Terminal
User		

Each object manager is named for the class of object it manages, so that the Ada object manager manages objects of class Ada and so on. The DDB object manager manages an object called the *dependency database*. The dependency database maintains a record of the dependencies between Ada units (for example, Ada *with* clauses). If users make changes to Ada units that affect dependencies, the dependency database is updated to reflect those changes. The DDB object manager removes obsolete dependencies from the database.

The Actions object manager manages the data structure that controls simultaneous access to objects. This data structure prevents inconsistencies when several users or jobs require simultaneous access to an object.

As part of their function, object managers compact the objects they manage. This part of their functionality is a *client* of the system daemon.

There are five clients in addition to the object managers:

Daily	Disk	Error_Log
Snapshot	Weekly	

The Daily client is a set of clients that are recommended to be run on a daily basis, as follows: Ada, DDB, Directory, Disk, Error_Log, and File.

The Disk client removes obsolete data from the disks. Running the Disk client is also called *disk collection*. Disk collection occurs both according to schedule and on an as-needed basis.

The Error_Log client periodically updates the Environment error log, which exists as a series of files in the world !Machine.Error_Logs. System errors and other messages can be directed to the operations console, the *stable-storage error log* on disk, or both with the Set_Log_Threshold procedure. Messages in the stable-storage error log are not accessible from the Environment until the Error_Log client copies the log into a permanent dated file in the world !Machine.Error_Logs.

The Snapshot client makes a record, or *snapshot*, of the current state of the Environment. Snapshots are important because, when the system boots, the Environment is restored to the state that was recorded in the most recent snapshot. Only objects that were committed before the last snapshot are preserved.

The Weekly client is a set of clients that are recommended to be run on a weekly basis, as follows: Archived_Code, Code_Segment, Configuration, Group, Link, Null_Device, Pipe, Session, Tape, Terminal, and User.

When Clients Run

Clients normally run automatically according to a default schedule established by the !Machine.Initialize procedure when the machine is booted. It is recommended that the schedule not be modified, except for the time at which the Daily and Weekly clients run, without consultation with a Rational technical representative.

Messages are displayed in the Message window, warning all users when a *major client* (Actions, Ada, DDB, Directory, Disk, File, or Snapshot) is about to run.

The system sends a warning message to all users two minutes before Daily and Weekly clients are run.

Clients can be removed from the schedule using the Quiesce procedure. Clients can also be run as needed (independently of the schedule) using the Run procedure.

When scheduling a client, keep in mind the effect of the client on the system. For example, snapshots preserve a consistent state of the Environment at an assigned moment in time. While a snapshot is running, all object managers suspend operation. On an active system supporting many users, this may affect the performance of users' jobs. Package Daemon includes utilities that broadcast information about upcoming snapshots, allowing users the opportunity to commit any unsaved files.

The Status procedure displays information on a given client (the default is Major_Clients), including its name, the next scheduled run, the most recent run, and the interval between runs. It also shows the current size of the client data structure, along with its size before and after the last run.

procedure Collect

```
procedure Collect (Vol      : Volume;
                   Priority : Collection_Priority := 0);
```

Description

Begins disk collection on the specified volume at the specified priority.

If scheduled disk collection is already in progress, the procedure returns immediately with no effect.

Note that the Collect procedure does not affect scheduling intervals.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Vol : Volume;`

Specifies the disk drive by disk drive number. The value 0 specifies all volumes.

`Priority : Collection_Priority := 0;`

Specifies the priority for running disk collection. Collection_Priority is an integer from −1 through 6. The default, 0, ensures that disk collection takes place, potentially affecting system load. The value −1 runs collection as a background activity; 6 allocates all resources for disk collection. The specified priority is temporary—that is, it affects only the disk collection run at the same time as the one you are about to start.

Example

The command:

```
   daemon.collect (2,0);
```

starts disk collection on disk drive 2, affecting system load.

subtype Collection_Priority

```
subtype Collection_Priority is Integer range -1 .. 6;
```

Description

Specifies the priority for running disk collection.

Enumerations

-1

Does not guarantee progress in disk collection. This priority runs disk collection as a very low-level background activity, using just "spare" CPU cycles. If there are none, disk collection will wait indefinitely (does a *backoff*) until cycles are available or until its priority is increased.

0

Guarantees progress in disk collection, with a small impact on system performance (and user response time). This priority is the same as running background jobs.

2

Guarantees progress in disk collection and preempts background jobs that do not use the best priority.

3

Guarantees progress in disk collection and runs on par with most foreground jobs. This priority has a big impact on system performance because it is sharing the same priority as most commands.

4

Guarantees progress in disk collections and preempts most foreground jobs. Editing is still possible, but commands will run very slowly.

6

Guarantees progress in disk collection and gives disk collection higher priority than user jobs.

type Condition_Class

```
type Condition_Class is (Normal, Warning, Problem, Fatal);
```

Description

Specifies how serious an error should be before it is logged in an Environment log file in the world !Machine.Error_Logs.

This type is used with the Set_Log_Threshold procedure.

Enumerations

`Fatal`

Specifies situations in which the Environment refuses to proceed. The task that was the source of the problem is suspended. Fatal messages appear in the error log preceded by the characters ***.

`Normal`

Specifies messages from the Environment that provide information but do not necessarily indicate problems. Normal messages appear in the error log preceded by the characters ---.

`Problem`

Specifies situations that the Environment expects will lead to problems. Problem messages appear in the error log preceded by the characters !!!.

`Warning`

Specifies situations that are unusual but not necessarily dangerous. Warning messages appear in the error log preceded by the characters +++.

References

procedure Set_Log_Threshold

function Get_Access_List_Compaction

```
function Get_Access_List_Compaction (Client : String := "") return Boolean;
```

Description

Returns true if any of the specified clients will be performing access-list compaction.

The clients that can perform access-list compaction are File, Ada, and Directory. Access-list compaction is the process of removing nonexistent groups from the access lists of objects. Nonexistent groups occur when groups are removed from the machine.

For further information on groups, see package Operator.

Enabling this feature slows a client's operation.

Parameters

```
Client : String := "";
```

Specifies the name of the client that will perform access-list compaction. Clients are listed in the introduction to this package.

```
return Boolean;
```

Returns true if any of the specified clients will be performing access-list compaction; otherwise, the function returns false.

References

procedure Set_Access_List_Compaction

package Operator

---

function Get_Consistency_Checking

```
function Get_Consistency_Checking (Client : String := "") return Boolean;
```

Description

Returns true if any of the specified clients will be performing consistency checking.

Consistency checking performs additional work to ensure that the internal state of the system is as it seems. This operation normally is run only when problems are suspected. For the DDB client, this setting may result in compaction.

Enabling this feature slows a client's operation.

Parameters

```
Client : String := "";
```

Specifies the name of the client to be queried to determine whether it is performing consistency checking. Clients are listed in the introduction to this package. The default specifies all clients.

```
return Boolean;
```

Returns true if any of the specified clients will be performing consistency checking. The default is false. If true, the default is restored after the next specified daemon run has completed.

References

procedure Set_Consistency_Checking

function Get_Log_Threshold

```
function Get_Log_Threshold (Kind : Log_Threshold) return Condition_Class;
```

Description

Returns the class of message that is handled according to the specified kind.

Parameters

```
Kind :  Log_Threshold;
```

Specifies the destination for messages. The destination can be Console_Print, Log_To_Disk, or Commit_Disk.

```
return Condition_Class;
```

Returns the class of message by Condition_Class type, including Normal, Warning, Problem, and Fatal.

procedure Get_Size

```
procedure Get_Size (Client                :     String;
                    Size                  : out Long_Integer;
                    Size_After_Last_Run   : out Long_Integer;
                    Size_Before_Last_Run  : out Long_Integer);
```

Description

Displays the current number of disk pages of the data structures compacted by the specified client, both before and after the last time the client was run.

This procedure is useful for monitoring the growth curve of the data structures that are compacted by the following clients: Ada, DDB, Directory, Disk, and File.

Parameters

`Client : String;`

Specifies the name of the client to be monitored. Clients are listed in the introduction to this package; however, the following clients are most relevant to this procedure: Ada, DDB, Directory, Disk, and File.

`Size : out Long_Integer;`

Specifies the number of pages currently used by the client's data structure.

`Size_After_Last_Run : out Long_Integer;`

Specifies the number of pages used by the client's data structure after the client was last run.

`Size_Before_Last_Run : out Long_Integer;`

Specifies the number of pages used by the client's data structure before the client was last run.

procedure Get_Snapshot_Settings

```
procedure Get_Snapshot_Settings (Warning        : out Duration;
                                 Start_Message  : out Boolean;
                                 Finish_Message : out Boolean);
```

Description

Returns the current snapshot options.

Parameters

`Warning :  out Duration;`

Specifies the current warning interval (the amount of time between the warning and the snapshot).

`Start_Message :  out Boolean;`

Specifies whether a message is displayed when the snapshot begins.

`Finish_Message :  out Boolean;`

Specifies whether a message is displayed when the snapshot is completed.

References

procedure Show_Snapshot_Settings

procedure Snapshot_Finish_Message

procedure Snapshot_Start_Message

procedure Snapshot_Warning_Message

PT, package Time_Utilities

function Get_Warning_Interval

```
function Get_Warning_Interval return Duration;
```

Description

Returns the amount of warning given before each Daily client runs.

The time between the warning message and the running of the client gives the operator time to quiesce it, if needed.

Parameters

```
return Duration;
```

Returns the amount of time between the warning and the running of the Daily client.

The value of Duration is a number of minutes, as defined in PT, package Time-_Utilities.

References

procedure Quiesce

PT, package Time_Utilities

function In_Progress

```
function In_Progress (Client : String) return Boolean;
```

Description

Returns whether the specified client is currently running.

Parameters

```
Client : String;
```

Specifies the name of the client in question. Clients are listed in the introduction to this package.

```
return Boolean;
```

Returns true when the specified client is running; otherwise, the function returns false.

function Interval

```
function Interval (Client : String) return Duration;
```

Description

Returns the scheduled interval for the specified client.

Parameters

```
Client : String;
```

Specifies the name of the client in question. Clients are listed in the introduction to this package.

```
return Duration;
```

Returns the current interval, in number of seconds.

References

PT, package Time_Utilities

function Last_Run

```
function Last_Run (Client : String) return Calendar.Time;
```

Description

Returns the last time the specified client was run.

Parameters

```
Client : String;
```

Specifies the client. Clients are listed in the introduction to this package.

```
return Calendar.Time;
```

Returns the time the client was last run.

type Log_Threshold

```
type Log_Threshold is (Console_Print, Log_To_Disk, Commit_Disk);
```

Description

Determines the actions that can be taken on a system message.

The action selected is controlled by the Set_Log_Threshold procedure, depending on the Condition_Class type of the message.

Enumerations

`Commit_Disk`

Writes messages to the stable-storage error log on disk immediately. These messages are retained if the system fails.

`Console_Print`

Directs messages to the operations console.

`Log_To_Disk`

Directs messages to the stable-storage error log on disk. These messages are not written permanently on the disk immediately, so they may be lost if the system fails before they can be written.

References

procedure Set_Log_Threshold

constant Major_Clients

```
Major_Clients : constant String := "*";
```

Description

Defines a constant string representing the list of major clients used as the default client in the Status procedure.

The list of clients includes: Actions, Ada, DDB, Directory, Disk, File, and Snapshot.

References

procedure Status

function Next_Scheduled

```
function Next_Scheduled (Client : String) return Calendar.Time;
```

Description

Returns the time at which the specified client will run.

Clients are listed in the introduction to this package.

Parameters

```
Client : String;
```

Specifies the name of the client in question. Clients are listed in the introduction to this package.

```
return Calendar.Time;
```

Specifies the next time the client will run.

procedure Quiesce

```
procedure Quiesce (Client           : String   := ">>CLIENT NAME<<";
                   Additional_Delay : Duration := 86_400.0;
                   Response         : String   := "<PROFILE>");
```

Description

Revokes the specified client's schedule and prevents it from running for the amount of time specified in the Additional_Delay parameter.

This procedure is equivalent to executing the Schedule procedure with a new value for the First_Run parameter and the same value for the Interval parameter.

Quiescing a daemon only prevents a client from running; it does not stop a client that is already running.

To prevent a client from running indefinitely, use Duration'Last for the Additional_Delay parameter.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Client : String := ">>CLIENT NAME<<";`

Specifies the name of the client to be delayed. Clients are listed in the introduction to this package. The default parameter placeholder ">>CLIENT NAME<<" must be replaced or an error will result.

`Additional_Delay : Duration := 86_400.0;`

Specifies the amount of delay. The default is one day. See PT, Time_Utilities.Seconds type.

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

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

Example

The command:

```
   daemon.quiesce ("error_log");
```

removes the Error_Log client from the schedule for one day.

References

procedure Schedule

PT, package Time_Utilities

procedure Run

```
procedure Run (Client   : String := "Snapshot";
               Response : String := "<PROFILE>");
```

Description

Runs the specified client immediately.

If the Snapshot client is specified, a snapshot is not taken until after the snapshot warning message interval has elapsed. (See the Snapshot_Warning_Message procedure.)

If the Disk client is specified, disk collection is done in the order of the disk with the least remaining space to the disk with the most remaining space. If the Status ("Disk") procedure is called while the disk collection is running, a volume whose space is yet to be collected has an asterisk (*) after its volume number.

Note that the Run procedure does not affect scheduling intervals nor does it schedule the client for additional runs.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Client : String := "Snapshot";`

Specifies the client to be run. Clients are listed in the introduction to this package. The default is the Snapshot client.

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

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

Example

The command:

```
daemon.run;
```

takes a snapshot after the Snapshot_Warning_Message procedure has elapsed. The client's scheduling intervals are not affected.

procedure Run
package !Commands.Daemon

References

procedure Snapshot_Warning_Message

---

procedure Schedule

```
procedure Schedule (Client     : String     := ">>CLIENT NAME<<";
                    Interval   : Duration;
                    First_Run  : Duration   := 0.0;
                    Response   : String     := "<PROFILE>");
```

Description

Schedules the specified client to run at regular intervals.

A Schedule procedure must be executed for each client. The Environment daemon cannot run a client unless it has been scheduled. The client schedules are set to default values, which the user can change.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Client : String := ">>CLIENT NAME<<";
```

Specifies the name of the client to be scheduled. Clients are listed in the introduction to this package. The default parameter placeholder ">>CLIENT NAME<<" must be replaced or an error will result.

```
Interval : Duration;
```

Specifies the period of time between runs. Duration'Last means the specified client will never run. The value of the Interval parameter is a number of seconds or an expression that evaluates to seconds. You can use the Minute, Hour, and Day constants from package !Tools.Time_Utilities, because they are of the Duration type.

```
First_Run : Duration := 0.0;
```

Specifies how soon a client runs after the Schedule procedure finishes executing. The value of the First_Run parameter is a number of seconds or an expression that evaluates to seconds. The default First_Run interval is 0.0 seconds; that is, the client will run immediately after invocation. You can use the Minute, Hour, and Day constants from package !Tools.Time_Utilities, because they are of the Duration type. In addition, the Time_Utilities.Duration_Until_Next function is useful because it returns the number of seconds between the time of execution and a specified time of day. This allows the schedule to be independent of the time at which the Schedule procedure is executed.

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

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

Example

The command:

```
    daemon.schedule ("ddb",86_400.0,36_000.0);
```

schedules the DDB (dependency database) client to run once a day, beginning 10 hours from the time the command was committed.

References

PT, package Time_Utilities

procedure Set_Access_List_Compaction

```
procedure Set_Access_List_Compaction (Client   : String  := "";
                                      On       : Boolean := True;
                                      Response : String  := "<PROFILE>");
```

Description

Specifies that access-list compaction should be performed by the specified clients.

The clients that can perform access-list compaction are File, Ada, and Directory. Access-list compaction is the process of removing nonexistent groups from the access lists of objects. Nonexistent groups occur when groups are removed from the machine. For further information on groups, see package Operator.

Enabling this feature slows a client's operation.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Client : String := "";
```

Specifies the name of the client. The default null string specifies all clients. The only clients that perform access-list compaction are File, Ada, and Directory. When all clients are specified, only those that can perform access-list compaction will actually do it.

```
On : Boolean := True;
```

Specifies whether access-list compaction should be turned on or off.

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

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

References

function Get_Access_List_Compaction

package Operator

procedure Set_Consistency_Checking

```
procedure Set_Consistency_Checking (Client   : String  := "";
                                    On       : Boolean := True;
                                    Response : String  := "<PROFILE>");
```

Description

Specifies whether consistency checking should be turned on.

Consistency checking performs additional work to ensure that the internal state of this system is as it seems. This operation normally is run only when problems are suspected.

Enabling this feature slows a client's operation.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Client : String := "";
```

Specifies the client that should perform consistency checking. Clients are listed in introduction to this package. The default null string specifies all clients.

```
On : Boolean := True;
```

Specifies whether consistency checking should be turned on or off.

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

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

References

function Get_Consistency_Checking

procedure Set_Log_Threshold

```
procedure Set_Log_Threshold (Kind  : Log_Threshold;
                             Level : Condition_Class);
```

Description

Specifies a destination for messages with a given level of severity.

By default, Condition_Class type messages of Warning, Problem, and Fatal are routed to the operations console. Messages of all kinds are written to the stable-storage error log on disk, causing the log to be committed to disk.

Parameters

`Kind : Log_Threshold;`

Specifies a destination for a class of messages. The destination can be Console_Print, Log_To_Disk, or Commit_Disk.

`Level : Condition_Class;`

Specifies the severity level of messages by Condition_Class type, including Normal, Warning, Problem, and Fatal.

Example 1

The command:

```
daemon.set_log_threshold (log_to_disk,normal);
```

routes all messages of condition class Normal (or greater) to the error log.

Example 2

The command:

```
daemon.set_log_threshold (console_print,problem);
```

routes all messages of condition classes Problem and Fatal to the operations console. Normal or Warning messages do not appear on the operations console.

procedure Set_Priority

```
procedure Set_Priority  (Priority : Collection_Priority := -1);
```

Description

Sets the priority for disk collection on the specified volume.

Executing this procedure while disk collection is in progress changes the priority of the current collection. If disk collection is not in progress when the command is executed, the procedure has no effect.

The Disk client runs at different priorities in response to a number of stimuli, as follows:

- Schedule procedure: Runs at priority 6.
- Run procedure: Runs at priority −1.
- Collect procedure: Runs at specified priority.
- Over threshold for the disk: Starts at priority 0 and escalates based on the number of disks that have reached the threshold.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Priority :  Collection_Priority := -1;
```

Specifies the priority for running disk collection. Collection_Priority is an integer from −1 through 6. The default, −1, runs collection as a background activity. The value 0 ensures that disk collection takes place, potentially affecting system load; 6 allocates all resources for disk collection.

Example

The command:

```
   daemon.set_priority (1,Ø);
```

sets the priority for disk collection to 0 on disk drive 1.

procedure Show_Log_Thresholds

```
procedure Show_Log_Thresholds;
```

Description

Displays current log thresholds in the current output window.

Example

The command:

```
daemon.show_log_thresholds;
```

displays information, such as the following, in the current output window:

```
Log Thresholds -- Console = WARNING, Logging = NORMAL, Commit = NORMAL
```

Console	Corresponds to log threshold Console_Print; shows that messages of Warning, Problem, and Fatal class are routed to the operations console.
Logging	Corresponds to log threshold Log_To_Disk; shows that messages of all class are written to the stable-storage error log on disk.
Commit	Corresponds to log threshold Commit_Disk; shows that messages of all class are written to the stable-storage error log, causing the log to be committed to disk.

procedure Show_Snapshot_Settings

```
procedure Show_Snapshot_Settings;
```

Description

Lists the current snapshot message options, showing the warning message interval and whether or not start and finish messages have been requested.

Example

The command:

```
   daemon.show_snapshot_settings;
```

returns a display as follows:

```
   Snapshot Settings -- Interval = 02:00.00, Start = TRUE, Finish = TRUE
```

The Environment issues a snapshot warning message two minutes before each snapshot and sends warning messages at the start and finish of each snapshot.

To see the interval between runs, use:

```
   daemon.status ("snapshot");
```

References

procedure Snapshot_Finish_Message

procedure Snapshot_Start_Message

procedure Snapshot_Warning_Message

procedure Status

procedure Snapshot_Finish_Message

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

Description

Tells the Environment whether or not to send a message informing users when a snapshot completes.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
On : Boolean := True;
```

Sends, when true, a message such as the following to users after every snapshot is completed:

```
   from SYSTEM: Ø2:34:34 PM; Snapshot has completed.
```

This message appears in the Message window.

When false, no finishing message is sent to users. The default is true.

Example

The command:

```
   daemon.snapshot_finish_message (false);
```

instructs the Environment not to notify users when a snapshot is completed.

procedure Snapshot_Start_Message

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

Description

Tells the Environment whether or not to send a message informing users when a snapshot begins.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`On : Boolean := True;`

Sends, when true, a message such as the following to users as each snapshot begins:

```
from SYSTEM: 02:31:55 PM;  Snapshot has started
```

This message appears in the Message window.

When false, no starting message is sent to users. The default is true.

Example

The command:

```
daemon.snapshot_start_message (false);
```

instructs the Environment not to notify users when a snapshot begins.

procedure Snapshot_Warning_Message

```
procedure Snapshot_Warning_Message (Interval : Duration := 120.0);
```

Description

Sends a warning message to users the specified number of seconds before the next snapshot begins.

When the interval is set to 0.0, no warning message is sent.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Interval : Duration := 120.0;
```

Specifies how soon, in seconds, to send a warning message such as the following before a snapshot begins:

```
from SYSTEM: 02:29:23 PM; Snapshot will start in 02:00.000
```

This message appears in the Message window.

The default is a 2-minute interval (120.0 seconds). When the Interval parameter is set to 0.0, no warning message is sent.

Example

The command:

```
daemon.snapshot_warning_message (300.0);
```

instructs the Environment to warn users of a pending snapshot 5 minutes (300.0 seconds) before it is run.

procedure Status

```
procedure Status (Client : String := "*");
```

Description

Displays (in the current output window) information on the current status for the specified client.

Parameters

```
Client : String := "*";
```

Specifies the client for which status is requested. The default ("*") is the Major_Clients constant. The null string ("") displays the status of all clients.

Example 1

The command:

```
daemon.status;
```

requests a display showing the status of Major_Clients (the default). The display includes the following categories of information:

CLIENT	The name of the client.
NEXT_TIME	The time of the client's next scheduled run.
PREVIOUS_TIME	The time of the client's most recent run.
INTERVAL	The interval of time between scheduled runs. The format for expressing intervals is as follows:

- mm:ss.ff indicates the number of minutes, seconds, and decimal fractions of seconds between runs. For example, the Actions client runs every 30 minutes, or 30:00.00.
- hh:mm:ss indicates the number of hours, minutes, and seconds between runs. For example, the Snapshot client runs every hour, or 01:00:00.
- dd/hh:mm indicates the number of days, hours, and minutes between runs. For example, the Ada client runs once a day, or 1/00:00.

SIZE	The current size, in pages, of the client's data structure.
POST	The size, in pages, of the client's data structure after the last run.
PRE	The size, in pages, of the client's data structure just before the last run.

The Status display typically looks like this:

```
Client      Next Time       Previous Time   Interval  Size    Post    Pre
==========  ==============  ==============  ========  ======  ======  ======
Actions     06/08/87 11:53  06/08/87 11:23  30:00.00      55      55      55
Ada         06/09/87 04:40  06/08/87 05:03  01/00:00    2073    2010    2425
Ddb         06/08/87 04:15  06/08/87 04:19  01/00:00   11213   10899   10899
Directory   06/08/87 04:30  06/08/87 04:36  01/00:00    5775    5698    6423
Disk        06/09/87 05:00  06/08/87 06:24  01/00:00  339000  311000  351000
File        06/08/87 04:45  06/08/87 05:14  01/00:00    1103     980    1219
Snapshot    06/08/87 12:08  06/08/87 11:10  01:00:00
```

Example 2

The command:

```
daemon.status ("disk");
```

checks the status of the client responsible for maintaining the disk data structure and returns a display such as the following:

```
Client  Next Time       Previous Time   Interval  Size    Post    Pre
======= ==============  ==============  ========  ======  ======  ======
Disk    06/09/87 05:00  06/08/87 06:24  01/00:00  339000  311000  351000
Vol 1   06/09/87 05:00  06/08/87 06:24  01/00:00  112000   95494  126000
Vol 2   06/09/87 05:00  06/08/87 06:24  01/00:00   79086   77639   81287
Vol 3   06/09/87 05:00  06/08/87 06:24  01/00:00   78126   71102   74536
Vol 4   06/09/87 05:00  06/08/87 06:24  01/00:00   69895   67360   69532
```

An asterisk before the volume number indicates that disk collection is running and that volume has not yet had collection run for it.

Example 3

The command:

```
daemon.status ("");
```

displays the status for all clients.

procedure Threshold_Warnings

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

Description

Specifies whether messages to all users currently logged into the system should be sent when collection thresholds have been passed.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
On :  Boolean := True;
```

Specifies whether messages should be sent. The default is true. False specifies that messages should not be sent.

subtype Volume

```
subtype Volume is Integer range 0 .. 31;
```

Description

Specifies a disk drive for procedures and functions.

The value 0 specifies all disk drives. (The value 0 should not be used in functions, because functions can return only a single value.)

procedure Warning_Interval

```
procedure Warning_Interval (Interval : Duration := 120.0);
```

Description

Sets the amount of warning time users are given before the Daily client runs.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Interval : Duration := 120.0;
```

Specifies how soon, in seconds, to send a warning message before the Daily client runs. The default is 2 minutes (120 seconds).

References

function Get_Warning_Interval

end Daemon;

package Message

This package contains two utilities for sending messages.

procedure Send

```
procedure Send (Who     : String;
                Message : String);
```

Description

Sends the message to the specified user.

The user must be logged in to receive the message. If the user is not logged in, the message is lost.

Messages appear in the Message window.

Parameters

`Who :  String;`

Specifies the username of the user to receive the message.

`Message :  String;`

Specifies the text of the message.

Example

The command:

```
   message.send ("wet","Who writes the daily messages?");
```

sends the string `From user:  Who writes the daily messages?` to user `WET`.

procedure Send_All

```
procedure Send_All (Message : String);
```

Description

Sends the message to all users.

Only users who are logged in receive the message.

Messages appear in the Message window.

Parameters

```
Message : String;
```
Specifies the text of the message.

Example

The command:

```
   message.send_all ("Who remembers how to abort a job?");
```

broadcasts the message `From user:  Who remembers how to abort a job?` to all users.

end Message;

RATIONAL

package Operator

The procedures described in package Operator are intended for system management tasks, including:

- Creating users
- Overseeing user sessions
- Managing groups for access control
- Enabling and disabling physical lines
- Performing operations without the restrictions of access control, when necessary

Commands such as the Change_Password and Create_Session procedures can be used by all users.

Access Control

Access to worlds, files, and Ada units and execution of certain operations is restricted by access control. Access control applies to all jobs, both those initiated when users directly execute commands and those explicitly initiated by users. A job's access to a world, file, or Ada unit is based on the username (also referred to as identity) of the user initiating the job. A username is granted access to an object if it is a member of one of the groups listed among the entries in the object's *access list* (ACL). Similarly, a job is granted access to an object based on the username of the user who initiates it. If the access list for a particular object does not contain an entry for a group to which the identity belongs, the job will not be permitted access to the object. For more information on access lists, see LM, introduction to package Access_List.

In addition to applying to certain objects, access control applies to certain operations within various packages of the Environment. Package Operator is one of these packages. Access control for these packages is documented in their corresponding sections of the *Rational Environment Reference Manual.*

Access Control and Groups

Group operations are performed by operations in package Operator. There are three types of groups: username groups, Environment-defined special groups, and user-defined groups. Each of these types of groups is described below.

Username Groups

When a username is created, a group name corresponding to that username is created by default. The username is, by default, a member of that group. Other users can also be made members of that group. For example, a username called Bill is created. A group named Bill is created by the Environment on creation of the username, and username Bill is a member of it. Another username, Sandy, is on the system. Sandy can be added to group Bill with the Add_To_Group procedure. This grants both usernames Bill and Sandy the specified access to any object that allowed access to group Bill.

Special Groups

Certain operations within the Environment can be performed only by usernames that meet one of the following requirements:

- The username is a member of group Operator.
- The username has write access to !Machine.Operator_Capability.
- The username is a member of group Privileged and is running in privileged mode.

These conditions can be broken into two types, as shown below.

Operator Capability

Members of group Operator (username Operator) and users with write access to file !Machine.Operator_Capability can perform operations within the Environment that require *operator capability.* Many of the operations in this package require operator capability.

Privileged Mode

Another special group, called Privileged, can use the Enable_Privileges procedure to turn on *privileged mode.* Privileged mode allows users to perform operations without the restrictions of access control. Privileged mode must be enabled in the same job as the operation(s) to be performed in privileged mode. Once the job has finished executing, privileged mode reverts to disabled.

Public and Network_Public Groups

Two predefined groups, called Public and Network_Public, are provided by the Environment. When a new username is created, the user automatically becomes a member of these two groups. Group Public can therefore be used to give everyone on a system access to a world, a file, or an Ada unit. In open shops, all worlds, files, and Ada units can include group Public on their access lists, in effect giving everyone access to everything on a system.

Group Network_Public is used in environments in which machines are using Rational Networking—TCP/IP. Inclusion of this group on access lists gives usernames access to objects on other machines on the same network.

Although usernames are added to these groups by default, they can be explicitly removed from them.

User-Defined Groups

Package Operator allows users with operator capability to create groups for access control. For example, rather than giving username Sandy access to group Bill as described in a previous example, a group can be defined for the two of them. This group is called Engineering. Both usernames Bill and Sandy are added to the group. Each username has its own group consisting of its username and each is also a member of group Engineering for access they want to share.

ACLs for New Username Home Worlds

When a new username is created, the access list (ACL) for the user's world is formed by concatenating the contents of !Machine.User_Acl_Suffix and *username*=>RCOD access, where *username* is the new username.

The default ACL for the world is formed by concatenating the contents of !Machine.User_Default_Acl_Suffix with *username*=>RW, where *username* is the new username.

Parameter Placeholders

Many of the commands in this package have, as a default, a parameter placeholder of the form ">>*name*<<". *Name* is the type of object that should replace >>*name*<<. Parameter placeholders must be replaced by the name of an object, as specified by their type. Executing a command containing a parameter placeholder will result in an error.

Response Profiles

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

procedure Add_To_Group

```
procedure Add_To_Group (User     : String := ">>USER NAME<<";
                         Group    : String := ">>GROUP NAME<<";
                         Response : String := "<PROFILE>");
```

Description

Adds the specified username to the specified group.

Execution of this procedure requires that the executing job have operator capability.

To see if the username is already a member of the group, you can use the Display_Group procedure. The username and group name must exist before this procedure is executed.

Note that identities are established at login. Adding or removing a user from a group will not be effective until the user's next login. Since a user's identity is established at login, the user must log out and then log back in before the new group membership is added to the user's identity.

Parameters

`User : String := ">>USER NAME<<";`

Specifies the username that should be added to the specified group. The default parameter placeholder ">>USER NAME<<" must be replaced or an error will result. The username must exist before this command is executed.

`Group : String := ">>GROUP NAME<<";`

Specifies the group name to which the username should be added. The default parameter placeholder ">>GROUP NAME<<" must be replaced or an error will result. The group name must exist before this command is executed.

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

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

Example

The command:

```
operator.add_to_group (user=>"bill",group=>"engineering");
```

adds username Bill to a group called Engineering.

Both username Bill and group Engineering must exist before this command is executed. Bill must log out and log back in again for his membership in group Engineering to be in effect.

References

procedure Create_Group

procedure Create_User

procedure Remove_From_Group

procedure Archive_On_Shutdown

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

Description

Specifies that certain data structures are archived in a representation-independent form whenever the system is shut down.

More specifically, this procedure stores the internal state of the object managers. Object managers include Actions, Ada, Archived_Code, Code_Segment, Configuration, DDB, Directory, File, Group, Link, Null_Device, Pipe, Session, Tape, Terminal, and User.

When these data structures are archived in representation-independent form, they can be restored even if the system is booted with a different release of the Environment.

Archiving causes the system to shut down and boot more slowly, so archiving is recommended only when required by Rational technical representatives for installing a new software release.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
On :  Boolean := True;
```

Specifies, when true, that archiving is included in the system shutdown process. The default is true.

procedure Cancel_Shutdown

```
procedure Cancel_Shutdown;
```

Description

Cancels a system shutdown initiated by the Shutdown procedure.

This procedure can be entered at any time during the interval before the actual shutdown takes place.

Execution of this procedure requires that the executing job have operator capability.

References

procedure Shutdown

procedure Change_Password

```
procedure Change_Password (User         : String := ">>USER NAME<<";
                           Old_Password : String := "";
                           New_Password : String := "";
                           Response     : String := "<PROFILE>");
```

Description

Replaces the Old_Password parameter with the New_Password parameter for the specified username.

Parameters

`User : String := ">>USER NAME<<";`

Specifies the name of the username. The default parameter placeholder ">>USER NAME<<" must be replaced or an error will result.

`Old_Password : String := "";`

Specifies the old password. If the old password is not known, the operator's password can be used (that is, the password for the username Operator). The default is the null string—in other words, no password.

`New_Password : String := "";`

Specifies the new password. The default is the null string—in other words, no password.

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

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

Example

The command:

```
   operator.change_password ("anderson","hello","greetings");
```

changes the password for user Anderson from `hello` to `greetings`.

procedure Create_Group

```
procedure Create_Group (Group    : String := ">>GROUP NAME<<";
                        Response : String := "<PROFILE>");
```

Description

Creates a new group with the specified name.

The group cannot already exist. When created, the group has no members. Members can be added with the Add_To_Group procedure.

A maximum of 1,000 group names is allowed per machine. Once this maximum has been reached, no further group names can be added. Groups that are no longer needed can be removed with the Delete_Group procedure, but deleting groups does not make it possible to create new groups once the limit has been reached. Access-list compaction must be run to make it possible to create new groups. See package Daemon for further information on access-list compaction.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Group : String := ">>GROUP NAME<<";`

Specifies the name of the group to be created. The default parameter placeholder `">>GROUP NAME<<"` must be replaced or an error will result.

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

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

Example

The command:

```
operator.create_group (group=>"ada_1_group");
```

creates a new group called Ada_1_Group. New members can be added to the group with the Add_To_Group procedure.

References

procedure Add_To_Group

procedure Delete_Group

procedure Remove_From_Group

---

procedure Create_Session

```
procedure Create_Session (User     : String := ">>USER NAME<<";
                          Session  : String := ">>SESSION NAME<<";
                          Response : String := "<PROFILE>");
```

Description

Creates another session for the specified user.

Parameters

`User : String := ">>USER NAME<<";`

Specifies the name of the user for whom a new session is to be added. The default parameter placeholder ">>USER NAME<<" must be replaced or an error will result.

`Session : String := ">>SESSION NAME<<";`

Specifies that the session name must be a legal Ada identifier, and no other object of this name should exist in the user's home world (!Users.User_Name). The default parameter placeholder ">>SESSION NAME<<" must be replaced or an error will result.

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

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

Example

The command:

```
    operator.create_session ("anderson","maintenance");
```

creates a new session called Maintenance for user Anderson.

procedure Create_User

```
procedure Create_User (User     : String  := ">>USER NAME<<";
                       Password : String  := "";
                       Volume   : Natural := 0;
                       Response : String  := "<PROFILE>");
```

Description

Opens a user account.

This procedure creates a username in !Machine.Users. In addition, the procedure creates the world !Users.User with the given Password parameter, if such a world does not already exist. The procedure also creates a default session, S_1, for the username. By default, the username is made a member of groups Public and Network_Public.

A group with the name specified by the User parameter is created and the new user is added to this group. Thus, each user has his or her own group with at least that user as a member.

When a new username is created, the access list for that world is formed by concatenating the contents of !Machine.User_Acl_Suffix and *username*=>RCOD, where *username* is the newly created username.

The new user is made a member of groups Public and Network_Public.

The default ACL for the world is formed by concatenating the contents of !Machine.User_Default_Acl_Suffix with *username*=>RW, where *username* is the newly created username.

Links from !Model.R1000 are copied into the new user's home world.

Sometimes, new accounts are assigned a temporary password. Use the Change_Password procedure to personalize passwords.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
User : String := ">>USER NAME<<";
```

Specifies the username. The name must be a legal Ada simple name and must be unique. The default parameter placeholder ">>USER NAME<<" must be replaced or an error will result.

```
Password : String := "";
```

Specifies the initial password. The password can be any arbitrary string. The default is the null string—in other words, no password.

```
Volume : Natural := 0;
```

Specifies the volume in which the user's home world will reside. The default, 0, lets the Environment choose the volume that has the most available space.

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

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

Example

The command:

```
operator.create_user ("anderson","andersonnew");
```

creates an account for user Anderson on a volume selected by the Environment. It assigns password andersonnew to the account. It also creates a home world for the user in !Users.Anderson, sets up the links for the world, sets the access list and default access list for the world, makes Anderson a member of groups Public and Network_Public, creates a group called Anderson, and adds Anderson to group Anderson.

References

procedure Change_Password

procedure Delete_Group

```
procedure Delete_Group (Group    : String := ">>GROUP NAME<<";
                        Response : String := "<PROFILE>");
```

Description

Deletes the group with the specified name.

This operation cannot be used to delete a group that has the same name as an existing username. When you execute the Delete_User procedure, it removes the group associated with that user. ACL entries that refer to a deleted group are reclaimed during the next access-list compaction. See package Daemon for further information on access-list compaction.

A maximum of 1,000 group names is allowed. Once this maximum has been reached, no further group names can be added. Groups that are no longer needed can be removed with the Delete_Group procedure and then reclaimed with the package Daemon access-list compaction.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Group :  String := ">>GROUP NAME<<";
```

Specifies the name of the group. The default parameter placeholder ">>GROUP NAME<<" must be replaced or an error will result.

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

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

Example

The command:

```
   operator.delete_group (group=>"ada_1_group");
```

deletes a group called Ada_1_Group.

procedure Delete_User

```
procedure Delete_User (User     : String := ">>USER NAME<<";
                       Response : String := "<PROFILE>");
```

Description

Disables login for the specified user but preserves the user's home world.

The procedure also deletes the user's entry from the !Machine.Users world and deletes the user's default session, S_1.

The user's home world can be deleted using commands in package !Commands.Library—for example, `library.destroy ("!users.user_name??");`.

Make sure that the user is logged out before disabling the user's account.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`User : String := ">>USER NAME<<";`

Specifies the username of the account to be disabled. The default parameter placeholder `">>USER NAME<<"` must be replaced or an error will result.

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

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

Example

The command:

```
   operator.delete_user ("anderson");
```

deletes the account for user Anderson and disables logins under Anderson's username. The world !Users.Anderson is preserved.

procedure Disable_Terminal

```
procedure Disable_Terminal (Physical_Line : Terminal.Port;
                            Response      : String        := "<PROFILE>");
```

Description

Disables the specified line for login.

If the line is in use, the command will not take effect until the user logs out.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Physical_Line : Terminal.Port;
```

Specifies that the terminal port is a number from 0 through 255.

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

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

Example

The command:

```
   operator.disable_terminal (18);
```

disables port 18.

References

procedure Enable_Terminal

procedure Disk_Space

```
procedure Disk_Space;
```

Description

Displays disk data.

Example

The command:

```
operator.disk_space;
```

returns a display such as the following:

```
Volume  Capacity  Available   Used    % Free
======  ========  =========  ======   ======
1        369120     284196    84924    76
2        391680     229239   162441    58
3        391680     274570   117110    70
4        391680     282619   118661    70

Total   1553760    1070624   483136    68
```

Volume indicates the disk drive. Capacity and Available describe disk space in terms of pages of 1 Kb each. Used describes the amount of disk space used in terms of pages of 1 Kb each. % Free specifies the amount of disk space that is still unused in terms of percentages.

The bottom row describes totals for all volumes.

procedure Display_Group

```
procedure Display_Group (Group    : String := ">>GROUP NAME<<";
                         Response : String := "<PROFILE>");
```

Description

Displays the usernames that are current members of the specified group on Io.Current_Output.

If there is a user with the name, it displays the groups of which the user is a member.

Parameters

`Group :  String := ">>GROUP NAME<<";`

Specifies the name of the group. The default parameter placeholder ">>GROUP NAME<<" must be replaced or an error will result.

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

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

Example

The command:

```
   operator.display_group (group=>"public");
```

results in the following display:

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

!USERS.GZC % OPERATOR.DISPLAY_GROUP                        STARTED 6:20:29 PM
------------------------------------------------------------------------------

Contents of group "public"
==========================
    user BILL
    user BLB
    user GZC
    user JIM
    user JMK
    user PUBLIC
    user SMP

User "public" is a member of
============================
   group PUBLIC
```

procedure Enable_Privileges

```
procedure Enable_Privileges (Enable : Boolean := True);
```

Description

Specifies that privileged mode should be enabled or disabled for the current job.

There is no effect on other jobs for that user or session, current or future.

For this procedure to execute successfully, the username must be a member of group Privileged. In general, privileged mode should not be enabled unless necessary to avoid accidentally doing something that normally would be restricted by access control.

When privileged mode is enabled, all tasks in that job become privileged. Execution of the procedure does not result in any output indicating that the username is now executing under privileged mode. Jobs spawned from a job with privileges enabled do not become privileged.

Privileged mode is enabled only for the duration of the job that called it. Therefore, it is not possible to enable it permanently for an entire session.

If the job is not a member of group Privileged, this command has no effect.

Parameters

```
Enable : Boolean := True;
```

Specifies, when true, that privileged mode should be enabled for the current username and session. When false, it specifies that privileged mode should be disabled. Thus, privileged mode can be enabled, disabled, and enabled again within the same job, if desired.

procedure Enable_Terminal

```
procedure Enable_Terminal (Physical_Line : Terminal.Port;
                           Response      : String        := "<PROFILE>");
```

Description

Enables the specified line for login.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Physical_Line : Terminal.Port;
```
Specifies that the terminal port is a number from 0 through 255.

```
Response : String := "<PROFILE>";
```
Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example

The command:

```
   operator.enable_terminal (18);
```

enables port 18.

References

procedure Disable_Terminal

procedure Explain_Crash

```
procedure Explain_Crash;
```

Description

Reads a shutdown cause and explanation from current input and enters them into the machine's error log.

The cause and explanation correspond to the information entered in the Shutdown procedure.

More specifically, this procedure is used to document system crashes or other service interruptions.

Input to this procedure is terminated with End_Of_Input.

procedure Force_Logoff

```
procedure Force_Logoff (Physical_Line   : Terminal.Port;
                        Commit_Buffers  : Boolean        := True;
                        Response        : String         := "<PROFILE>");
```

Description

Terminates any session active on the specified line.

Uncommitted changes to images are saved if the Commit_Buffers parameter is true. The user's background jobs (if any) continue to run, and any foreground jobs that do not require interactive input are put in the background. Foreground jobs that attempt interactive input are killed.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Physical_Line : Terminal.Port;
```

Specifies that the terminal port is a number from 0 through 255. You can determine which port by executing the !Commands.What.Users procedure.

```
Commit_Buffers : Boolean := True;
```

Specifies whether uncommitted changes the user has made to any images will be committed. When true (the default), the changes are saved.

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

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

Example

The command:

```
   operator.force_logoff (18);
```

logs off the user currently on port 18. Any uncommitted changes to images are saved.

function Get_Archive_On_Shutdown

```
function Get_Archive_On_Shutdown return Boolean;
```

Description

Returns a Boolean indicating whether archiving on shutdown has been enabled by the Archive_On_Shutdown procedure.

Parameters

```
return Boolean;
```

Specifies, when true, that system shutdown archives object managers.

References

procedure Archive_On_Shutdown

function Get_Login_Limit

```
function Get_Login_Limit return Positive;
```

Description

Returns the maximum number of users that can be logged in at one time on a machine.

The maximum number can be set by the Limit_Login procedure.

Parameters

```
return Positive;
```

Returns the number of concurrent logins.

References

procedure Limit_Login

function Get_Shutdown_Interval

```
function Get_Shutdown_Interval return Duration;
```

Description

Returns the current interval that is used by the Shutdown procedure.

The interval is set by the Shutdown_Warning procedure.

Parameters

```
return Duration;
```

Returns the number of seconds between entering the Shutdown procedure and the actual shutting down of the system.

References

procedure Shutdown

procedure Shutdown_Warning

---

procedure Internal_System_Diagnosis

```
procedure Internal_System_Diagnosis;
```

Description

Runs the Environment Elaborator Database (EEDB) from an Environment window rather than on the operations console.

Execution of this procedure requires that the executing job have operator capability.

procedure Limit_Login

```
procedure Limit_Login (Sessions : Positive := Positive'Last);
```

Description

Sets a limit on the number of concurrent user logins for a machine.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Sessions :  Positive := Positive'Last;
```

Specifies the maximum number of logins. The default is Positive'Last.

Example

The command:

```
operator.limit_login (16);
```

limits the number of login sessions to 16.

function Privileged_Mode

```
function Privileged_Mode return Boolean;
```

Description

Returns true if privileged mode is enabled for the calling job.

Parameters

```
return Boolean;
```

Returns true if privileged mode is enabled, or false if it is disabled, for the calling job.

procedure Remove_From_Group

```
procedure Remove_From_Group (User     : String := ">>USER NAME<<";
                             Group    : String := ">>GROUP NAME<<";
                             Response : String := "<PROFILE>");
```

Description

Removes the specified username from the specified group.

To determine whether the username is a member of the group before removing it, use the Display_Group procedure.

Note that identities are established at login. Removing a user from a group will not be effective until the user's next login.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`User : String := ">>USER NAME<<";`

Specifies the username that should be removed from the specified group. The default parameter placeholder ">>USER NAME<<" must be replaced or an error will result.

`Group : String := ">>GROUP NAME<<";`

Specifies the group name from which the username should be removed. The default parameter placeholder ">>GROUP NAME<<" must be replaced or an error will result.

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

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

Example

The command:

```
operator.remove_from_group (user=>"bill",group=>"engineering");
```

removes username Bill from the group called Engineering.

References

procedure Add_To_Group

procedure Create_Group

procedure Create_User

procedure Display_Group

procedure Set_System_Time

```
procedure Set_System_Time (To_Be    : String := ">>TIME<<";
                           Response : String := "<PROFILE>");
```

Description

Resets the system clock, for example, for changing to or from daylight savings time.

Because the system clock has an independent power supply (a battery), there is no need to reset the clock if the system is powered down.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`To_Be : String := ">>TIME<<";`

Specifies a date, time, or combination of date and time expressed in one of the formats listed below. The To_Be parameter consists of 2 through 6 two-digit numbers delimited by nonnumeric characters. The default parameter placeholder ">>TIME<<" must be replaced or an error will result.

In general, the numbers can be interpreted, because each component of a date or time has its own range (for example, 85 is always interpreted as a year, because it cannot be anything else).

In the following examples of allowable times, YY expresses a year, MM expresses a month, DD expresses a day, HH expresses an hour, mm expresses minutes, and SS expresses seconds.

YY/MM/DD HH:mm:SS	MM/DD HH:mm:SS
MM/DD/YY HH:mm:SS	MM/DD HH:mm
MM/DD/19YY HH:mm:SS	YY/MM/DD
YY/MM/DD HH:mm	MM/DD/YY
MM/DD/YY HH:mm	HH:mm:SS
	HH:mm

Other allowable time and date formats are described in PT, Time_Utilities.Time_Format and Time_Utilities.Date_Formattypes.

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

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

Example 1

The command:

```
   operator.set_system_time ("15:00");
```

sets the time to 3:00 P.M. today.

Example 2

The command:

```
   operator.set_system_time ("06/12/87");
```

changes the date to June 12 without changing the time.

---

procedure Show_Login_Limit

```
procedure Show_Login_Limit;
```

Description

Displays the maximum number of concurrent logins on the current output.

The maximum number of logins can be set by the Limit_Login procedure.

Example

The command:

```
    operator.show_login_limit;
```

displays the following when the login limit is 16:

```
    It is currently possible for 16 users to be logged in
```

References

procedure Limit_Login

procedure Show_Shutdown_Settings

```
procedure Show_Shutdown_Settings;
```

Description

Displays the shutdown settings on the current output.

Shutdown settings are set by the Shutdown_Warning and Archive_On_Shutdown procedures.

Example

The command:

```
   operator.show_shutdown_settings;
```

produces a message such as the following on the current output:

```
   Shutdown Interval is 01:00:00; Archive_Enabled = False
```

References

procedure Archive_On_Shutdown

procedure Shutdown_Warning

procedure Shutdown

```
procedure Shutdown (Reason      : String := "COPS";
                    Explanation : String := "Cause not entered");
```

Description

Shuts down the system after the interval set by the Shutdown_Warning procedure has passed.

The shutdown cause is specified by the Reason parameter, and the explanation is specified by the Explanation parameter. These are entered into the machine's error log.

The Shutdown procedure issues several warnings to users. The first warning occurs when the procedure is executed, the next occurs after 3/4 of the interval has passed, the next occurs when 3/4 of the remaining time has passed, and so on, until the system is shut down. Note that a warning interval of 30 seconds or less results in immediate shutdown.

When the system is shut down, users are logged off, all terminal lines are disabled, and a snapshot is taken to preserve the Environment state.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Reason : String := "COPS";`

Specifies a reason for shutting down the machine. The default, "COPS", specifies a customer shutdown. The shutdown does not happen unless the reason is a valid one. The value "?" for this parameter gives a list of valid reasons and does not perform the shutdown. The cause is entered into the machine's error log.

`Explanation : String := "Cause not entered";`

Specifies an explanation to be entered into the machine's error log. The default is "Cause not entered" and should be replaced.

References

procedure Cancel_Shutdown

procedure Explain_Crash

procedure Shutdown_Warning

---

procedure Shutdown_Warning

```
procedure Shutdown_Warning (Interval : Duration := 3600.0);
```

Description

Sets the interval between the time the Shutdown procedure is executed and the time the system actually shuts down.

The Interval parameter also determines when the shutdown warning messages are issued. The first message is sent at the beginning of the interval, the second is sent after 3/4 of the interval has passed, and so on, until the system shuts down.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Interval : Duration := 3600.0;
```

Specifies that the default for the interval is 3,600 seconds, or 1 hour. The Interval parameter is rounded to the nearest minute. Less than 1 minute is rounded to 0.

Example

The command:

```
   operator.shutdown_warning (1800.0);
```

sets the interval to 30 minutes. The next time the Shutdown procedure is executed, 30 minutes will pass before the system actually shuts down.

end Operator;

package Queue

Package Queue contains procedures for printing files and for creating and managing print queues, both on the user's current machine and on other machines through Rational Networking—TCP/IP.

Some of the functionality available in this package can also be accessed through the username's session switch file. For further information on switch files, see LM, package Switches.

If your system has Rational Networking—TCP/IP, you can submit print requests to other machines on the same network and query other machines on the same network.

Some of the commands in this package take the Options parameter. For further information on the syntax of Options parameters, see Key Concepts.

The Print procedure can be used to submit one or more objects for printing. This procedure submits objects to the default print queue class (the class used, by default, when a class is not specified). The user can request another class by specifying a nondefault value for the Class parameter. By default, the system notifies the user in the Message window when the request is complete. The user can select options for notification through the message utility or select not to be notified (Notify option).

The Print procedure automatically prints the user's name on a separate banner page, prints the object name and the page number on each page, and wraps lines longer than 80 characters. The Print procedure also permits the user to specify a different banner, headers, line length, and other format characteristics.

When a user makes a print request:

1. The object to be printed is submitted to the *print spooler*, which performs basic formatting such as inserting page breaks, headers, and footers.
2. The print spooler then queues the object to the *class* specified by the user in the print request. (A class is a logical grouping of print requests that allows these requests to be handled as a set.) If no class is specified, the default class is used. An object submitted for printing is queued along with any other objects submitted to the same class.

3. Because each class is typically associated (or *registered*) with at least one *device*, the class to which an object is submitted determines the device that will handle the print request. A device can be a physical device (such as a printer) or a logical device (such as a file).
4. If the designated device is enabled, the submitted object is printed. If the device has been disabled, the submitted object remains queued until the device is enabled (or the class is reregistered with some other enabled device).

The print spooler maintains the relationships among classes and devices. The term *print queue* refers to any class that is registered with a device. Put another way, print requests are *queued* to a class, where they wait to be handled by an associated device.

Execution of some of the operations in this package requires that the job executing the commands have operator capability. These operations include Add, Create, Destroy, Disable, Enable, Kill_Print_Spooler, Register, Remove, Restart_Print-_Spooler, and Unregister.

Users with operator capability can create any number of classes for queuing print requests. For example, one class can be designated for long batch jobs and another class for short urgent jobs, associating each of these classes with its own device. Similarly, a separate class can be assigned to each user group or department so that requests from certain groups can be routed to specific devices, suspended, or given preference as needed.

Note that a single device can have more than one class registered with it—for example, when several departments use the same printer. Furthermore, a single class can be registered with more than one device. A print request submitted to such a class is routed to the first available device.

Users with operator capability described above can create a print queue by following the procedure described below:

1. Use the Restart_Print_Spooler procedure to ensure that the print spooler is running.
2. Use the Create procedure to define a class with the specified name.
3. Use the Add procedure to add a device with the specified name.
4. Use the Register procedure to associate the class with a device.
5. Use the Default procedure to define a default class with the specified name.
6. Use the Enable procedure to make the device available for use.

procedure Add

```
procedure Add (Device  : String := "";
               Options : String := "XON_XOFF");
```

Description

Specifies a new device to be added to the system with the specified device name.

This procedure allows you to add a new device and to specify the kind of protocol required between print jobs (if any). Devices must be added before they can be registered using the Register procedure.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Device : String := "";
```

Specifies the device by physical line number. The line number takes the form ***terminal_n***, where ***n*** is the number of the port to which the device is attached.

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

Specifies the type of protocol required by the device to be added. For information on using Options parameters, see Key Concepts. The default is XON_XOFF. This parameter also specifies a Telnet host to be used and the socket, if desired.

`XON_XOFF` Boolean

Specifies standard flow control.

`RTS` Boolean

Specifies standard flow control (XON_XOFF), with RTS protocol used between print requests. This option is used when two machines are sharing a common device to resolve connection problems.

`DTR` Boolean

Not used currently. Reserved for future development.

`Host=>`***name***

Specifies standard flow control (XON_XOFF), with a Telnet connection used between print requests, where ***name*** is the name of the Telnet connection to be used.

`Socket=>`***socket number***

If a Telnet connection is specified with the Host option, the Socket option can also be specified—for example, `"host=>lab_print,socket=>(0,23)"`.

Example 1

The command:

```
queue.add ("terminal_21");
```

adds a device called Terminal_21.

Example 2

The command:

```
queue.add (device=>"terminal_255",options=>"host=>lab_print,socket=>(0,23)");
```

adds a Telnet device called Terminal_255.

References

procedure Enable

procedure Register

procedure Remove

constant All_Classes

```
All_Classes : constant Class_Name := "all";
```

Description

Defines a constant that represents all defined classes.

constant All_Spooler_Devices

```
All_Spooler_Devices : constant String := "all";
```

Description

Defines a constant that represents all devices registered with at least one class.

---

procedure Cancel

```
procedure Cancel (Request_Id : Positive);
```

Description

Cancels the specified print request.

This procedure cancels requests whether or not those requests have started to print. The value of the Request_Id parameter can be obtained with the Display procedure.

Although the cancel request will complete quickly, the actual canceling can take several minutes before the print spooler removes/terminates the request.

Parameters

```
Request_Id : Positive;
```

Specifies the number assigned to the print request.

References

procedure Display

procedure Classes

```
procedure Classes (Which        : Class_Name  := "all";
                   Show_Devices : Boolean     := True);
```

Description

Displays information about the specified classes.

Parameters

`Which : Class_Name := "all";`

Specifies the class for which information is requested. The default is all classes. Users on installations that use Rational Networking—TCP/IP to connect multiple R1000 systems can query other machines on the network. Thus, the name can specify a machine name of the form !!*machine name*, where *machine name* is the name of a machine—for example, !!M1.

`Show_Devices : Boolean := True;`

Specifies whether to display information on devices as well as on classes. The default is true.

Example

The command:

```
   queue.classes;
```

produces a display such as the following:

```
     Class        Device(s)
    =======  ====================
     LP       TERMINAL_34
     PL       TERMINAL_250
```

This display shows that class LP is associated with the device Terminal_34. That is, print requests made to LP are routed to Terminal_34. It also shows that class PL is associated with device Terminal_250, which is, by convention, a Telnet port.

subtype Class_Name

```
subtype Class_Name is String;
```

Description

Defines the form of a name assigned to a given set of devices.

All class names are mapped to uppercase (that is, case-insensitive). This subtype can contain the name of a machine, using the format !!*machine_name*, where *machine_name* is the name of another machine on the same network—for example, !!M1.LP. The remote machine name can be used only to query and print on remote machines. It cannot be used to change the print spooler configuration on remote machines.

procedure Create

```
procedure Create (Class : Class_Name := "");
```

Description

Creates a class with the specified name.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Class : Class_Name := "";
```

Specifies the case-insensitive name for the class being created. The default is no name. You cannot use a remote machine name to create a class on a remote machine.

Example

The command:

```
queue.create ("lp");
```

creates a class called LP.

References

procedure Destroy

procedure Register

procedure Default

```
procedure Default (Class : Class_Name := "");
```

Description

Specifies a new default class for all Environment print requests.

The procedure sets the default class to the specified class and prints a message in the Message window. If the default parameter value is used, the procedure simply prints the name of the current default class in the Message window.

The class must exist before it can be made the default class. Use the Create procedure to create classes.

You can also assign a remote class as the default class, so that the default causes jobs to be queued on a remote machine.

Execution of this procedure requires that the executing job have operator capability. Other users can execute this procedure to display the default class in the Message window.

Parameters

```
Class : Class_Name := "";
```

Specifies a new default class name. If the default value of this parameter is used, the only effect of this command is to display the current default class. Users on installations that use Rational Networking—TCP/IP to connect multiple R1000 systems can query other machines on the network. Thus, a remote machine name can be specified to query a remote machine for its default class—for example, !!M1 is a machine name. You cannot set the default class on a remote machine.

Example

The command:

```
   queue.default ("newclass");
```

specifies Newclass as the default class for all print requests.

References

procedure Create

---

procedure Destroy

```
procedure Destroy (Class   : Class_Name := "";
                   Reroute : Class_Name := "");
```

Description

Removes the specified class and routes existing requests for that class to another class.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Class : Class_Name := "";`

Specifies the name of the class to be removed. The default is no name.

`Reroute : Class_Name := "";`

Specifies a class for routing. Print requests spooled to the removed class are routed to this class. The default is to route requests to the default class. A remote machine name cannot be specified to destroy a class on a remote machine.

Example 1

The command:

```
queue.destroy ("lp","newclass");
```

removes LP from the list of active classes and routes all requests for LP to Newclass.

Example 2

The command:

```
queue.destroy ("lpr3");
```

removes LPR3 from the list of active classes and routes all requests for LPR3 to the default class.

References

procedure Destroy

---

procedure Devices

```
procedure Devices (Which        : String  := "all";
                   Show_State   : Boolean := True;
                   Show_Classes : Boolean := True);
```

Description

Displays information about the specified devices.

Parameters

```
Which : String := "all";
```

Specifies the device for which information is requested. The default is to show information about all devices.

```
Show_State : Boolean := True;
```

Requests information on the current state of the devices, whether enabled or disabled. The default is true.

```
Show_Classes : Boolean := True;
```

Specifies whether to display information on classes associated with the displayed devices. The default is true.

Example

The command:

```
queue.devices;
```

produces a listing such as the following:

```
  Device      Protocol  Characteristics  State     Classes
===========  ========  ===============  ========  =======
TERMINAL_40  XON_XOFF  Laser_Comm       Disabled  (none)
TERMINAL_32  RTS                        Disabled  (none)
TERMINAL_255 TELNET                     Enabled   LP
             (postscript
              (0,23))
```

This display shows three devices. Two are disabled and have no associated classes. Terminal_255, however, is enabled and is associated with class LP.

procedure Disable

```
procedure Disable (Device    : String  := "";
                   Immediate : Boolean := False);
```

Description

Disables the specified device.

Depending on the value of the Immediate parameter, the procedure disables the specified device either before or after the current print request has finished on that device. If the Immediate parameter is false (the default), the procedure waits until the current print request has completed. If Immediate is true, the device is disabled immediately and the interrupted print request is placed back on the print queue to be reprinted when possible.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Device :  String := "";
```

Specifies the device by physical line number. The line number takes the form *terminal_n*, where *n* is the number of the port to which the device is attached.

```
Immediate :  Boolean := False;
```

Specifies whether to allow the current print request to finish before the device is disabled. If the Immediate parameter is false (the default), the procedure waits until the current print request has completed. If Immediate is true, the device is disabled immediately and the interrupted print request is placed back on the print queue to be reprinted when possible. The Disable procedure will return quickly; however, it can take several minutes before the interrupted job is requeued and the device disabled.

Example 1

The command:

```
   queue.disable ("terminal_40",true);
```

disables the device on physical line 40. Because of the parameter true, the device is disabled immediately. Any print request that is actually printing when this command is executed is placed on the print queue again.

Example 2

The command:

```
   queue.disable ("terminal_40");
```

disables the device on physical line 40. Because of the default parameter false, the print request currently on this device must complete before the device is disabled.

References

procedure Add

procedure Enable

---

procedure Display

```
procedure Display (Class : Class_Name := "all");
```

Description

Displays the print requests currently queued in the specified class.

The display appears in the current output window. If there are no queued requests, a message to this effect appears in the Message window.

The display shows the identification number for each request. Use the appropriate number as the Request_Id parameter when using the Cancel procedure.

Parameters

```
Class : Class_Name := "all";
```

Specifies the class for which the contents are to be displayed. The default is to display the contents of all classes. Users on installations that use Rational Networking—TCP/IP to connect multiple R1000 systems can query other machines on the network. Thus, the name can specify a machine name of the form !!*machine name*, where *machine name* is the name of a machine—for example, !!M1.

Example

The command:

```
   queue.display;
```

produces a display such as the following in the current output window:

```
   ID   Time   State  Class    User        Object
   ==  =====  ======  =====  ========  ==================
   20  17:54  Queued   LP    OPERATOR  !USERS.OPERATOR.LOG
```

In this example, the state of the print request is queued because the device associated with the class LP is disabled. When a print request is currently being processed by an enabled device, the state of the request is active.

References

procedure Cancel

---

procedure Enable

```
procedure Enable (Device : String := "all");
```

Description

Enables the specified device.

The default is to enable All_Spooler_Devices (all registered devices).

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Device :  String := "all";
```

Specifies the device by physical line number. The line number takes the form *terminal_n*, where *n* is the number of the port to which the device is attached. The default is all devices.

Example

The command:

```
   queue.enable ("terminal_40");
```

enables the spooler device on physical line 40. Note, however, that line 40 cannot be simultaneously enabled for login.

References

procedure Add

procedure Disable

procedure Kill_Print_Spooler

```
procedure Kill_Print_Spooler;
```

Description

Stops the print spooler.

This procedure disables all spooler devices, prevents additional print requests, and prevents the successful completion of any queue operations requiring this machine. The print spooler can be restarted with the Restart_Print_Spooler procedure.

Execution of this procedure requires that the executing job have operator capability.

References

procedure Restart_Print_Spooler

procedure Print

```
procedure Print (Name    : String := "<IMAGE>";
                 Options : String := "<DEFAULT>";
                 Banner  : String := "<DEFAULT>";
                 Header  : String := "<DEFAULT>";
                 Footer  : String := "<DEFAULT>");
```

Description

Queues the specified objects for printing.

You can specify one or more objects by naming, by selection, or by placing the cursor in a window containing the object's images.

Besides text files, you can get printouts of Ada units and library listings. To print images from output windows or images of other kinds of objects (for example, a switch file), you must first copy the object's image into a text file, commit the file, and then print that text file.

The default is to queue the print request to the device associated with the default class and to notify when the jobs are complete. The Message window echoes all print requests, unless the switches are set differently in the user's switch file.

The Print procedure uses the Options parameter to allow you to change the printout format or to request multiple copies of your print request.

Parameters

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

Specifies the name of the object to be sent to the print queue. This parameter can use special names and wildcards to specify a set of objects. The default is the current selection or image.

You can specify a text file, an Ada unit, or a library (which prints a list of the library's contents). If you specify a file or an Ada unit using the current image, the most recently committed version is printed. Therefore, the printout will differ from the actual image on the screen if that image contains uncommitted changes.

`Options : String := "<DEFAULT>";`

Specifies the options to be used in formatting output. The following is a list of the options available for use in the Options parameter to format output. Note that the Options parameter uses the special name "<DEFAULT>". When this special name is used, the system looks in the session switch file for the options set in the Queue.Options switch. If the switch file is not accessible, the system uses the options "FORMAT=>(Wrap, System_Header)". These options are no longer true and must be respecified if any of the options are changed by substituting an option for the "<DEFAULT>" special name. One of the following three options must be specified: Original_Raw, Raw, or Format. Unless otherwise specified in the Options parameter, the Boolean options Original_Raw, Raw, and Spool_Each_Item are false. The other options take the defaults specified below.

Banner_Page_User_Text=*string*

Specifies a string (with a maximum of 60 characters) that will be printed on the banner page, beneath the banner and above the system-generated information.

Class=*string*

Specifies the name of the class, where *string* is the class to which the print request is queued. The class determines the device that will handle the print request. Note that the specified class must exist and must be associated with an enable device. If this option is not specified, the class is the default class.

Copies=*positive integer*

Specifies the number of copies to be printed, where *positive integer* is the number of copies to be printed. Copies are generated one at a time, and other jobs may intervene between copies. The default is 1 copy.

Format options

The Format option is an options parameter within the Options parameter that can be used to specify format options Wrap, Truncate, Numbering, System_Header, Width, Length, and Tab_Width. Options must appear in parentheses—for example, FORMAT=>(Wrap, Width=77).

Length=*positive integer*

Specifies the total number of printed lines per page, including headers and footers.

The default length is 60 lines. Note that, by default, the Rational Printer is set to eject a page after 66 lines if a formfeed is not encountered earlier. To print pages longer than 66 lines, you must change the appropriate setting on the Rational Printer (see "Printer Operations and Maintenance" in the *Rational R1000 Development System: System Manager's Guide*) in addition to increasing the Length value.

The number of lines in the body text for each page is automatically adjusted to accommodate any combination of a one-line header, a one-line footer, or a system page header. However, if you specify a multiple-line header or footer, you must decrease the Length value for every additional line beyond the expected one.

Numbering Boolean

Specifies whether to provide line numbering. The default is false.

System_Header Boolean

Specifies whether to print the system page header on each page. The system header is the name of the object and a page number. If there is a user-specified header, the system page header appears above it. The default is false.

Tab Width=*positive integer*

Specifies number of spaces with which to replace a Tab character (Ascii.Ht). The default is 8. A value of 0 specifies no replacement.

Truncate Boolean

Specifies whether to truncate lines that are longer than Width. The default is false. If both Truncate and Wrap are set to true, Wrap is assumed to be true.

Width=*positive integer*

Specifies the maximum number of printable characters per line, where *positive integer* is the number of characters.

The default width is 80 columns. Note that the Rational Printer itself can be set to wrap after 80 columns. To print wider pages, you must change the appropriate setting on the Rational Printer (see "Printer Operations and Maintenance" in the *Rational R1000 Development System: System Manager's Guide*) in addition to increasing the value of Width.

The Wrap and Truncate options specify what to do with lines that are longer than Width.

Wrap Boolean

Specifies whether to wrap lines that are longer than Width. The default is false. If both Truncate and Wrap are set to true, Wrap is assumed to be true. The wrapped portions of wrapped lines do not receive a new line number.

Original_Raw Boolean

For use on machines low on space when large files need to be printed. This option prints without using space to make the print spooler copy. The file can be spooled only to a local device. Each file is spooled separately (that is, it ignores the Spool_Each_Item option). A message is sent when printing is complete (that is, it ignores the Notify option). A banner page is printed (that is, it ignores the Banner_Page_User option).

Notify literal

Specifies the manner of notification after a print request is completed. By default, an informative message is sent to the Message window. The available types are None, Message (the default), and Mail (reserved for future development). Remote requests, under normal conditions, will also notify you.

Raw Boolean

Specifies whether the printer should interpret the input. Prints the file without interpreting characters (that is, without recognizing formfeeds or linefeeds). This is useful for preformatted text or binary data. Using this option turns off other options. It does not provide a system or user header.

Spool_Each_Item Boolean

Specifies whether to spool each file indicated by the Name parameter as a separate job. When true, each file has its own banner page. When false, a single banner page is printed. The default is false.

```
Banner : String := "<DEFAULT>";
```

Specifies the string that appears on the single banner page that precedes the printout. The string you supply is truncated at 11 characters. If the null string is specified, a banner page will not be generated.

The special name "<DEFAULT>" refers to the banner that is specified in the username's session switch file or the username if one is not specified.

```
Header : String := "<DEFAULT>";
```

Specifies a line of text that appears at the top of each page of the printout. Any nonnull string (including blank characters) constitutes a user-specified header. A blank line is automatically inserted below the user-specified header to separate the header text from the printout.

If the Options parameter requests a system page header in addition to the user-specified header, then the system header appears first, followed by the user-specified header.

The user-specified header can be longer than the Width option; however, a lengthy header is not wrapped automatically. You must include linefeeds in a header if you want it to wrap onto multiple lines.

The number of lines in the body text for each page is automatically adjusted to accommodate any combination of a one-line header, a one-line footer, or a system page header. However, if you specify a multiple-line header, you must decrease the Length value for every header line beyond the expected one.

The special name "<DEFAULT>" refers to the header that is specified in the username's session switch file.

```
Footer : String := "<DEFAULT>";
```

Specifies a line of text that appears at the bottom of each page of the printout. Any nonnull string (including blank characters) constitutes a user-specified footer. A blank line is automatically inserted above the user-specified footer to separate the footer text from the printout.

The user-specified footer can be longer than the Width option; however, a lengthy footer is not wrapped automatically. You must include linefeeds in a footer if you want it to wrap onto multiple lines.

The number of lines in the body text for each page is automatically adjusted to accommodate any combination of a one-line header, a one-line footer, or a system page header. However, if you specify a multiple-line footer, you must decrease the Length value for every footer line beyond the expected one.

The special name `"<DEFAULT>"` refers to the footer that is specified in the username's session switch file.

Example

The command:

```
queue.print (name=>"output_samples",options=>"copies=2,truncate,
                    system_header",banner=>"dept 04",
                    header=>"May 9, 1987");
```

prints two copies of the object Output_Samples, with Dept 04 on the banner page and the date appearing under the system page header. Lines longer than 80 characters are truncated.

The request to print Output_Samples is queued to the default class, and a message such as the following appears in the Message window:

```
Request number 58 has been queued
```

A further message in the Message window notifies the user when the job is complete. The user would also be notified if the request were made on a remote machine.

References

SJM, Session Switches

procedure Print_Version

```
procedure Print_Version (The_Version  : Directory.Version;
                         Options      : String := "<DEFAULT>";
                         Banner       : String := "<DEFAULT>";
                         Header       : String := "<DEFAULT>";
                         Footer       : String := "<DEFAULT>");
```

Description

Queues the specified object version for printing, allowing customization of the printout page format.

The default is to assume the device associated with the default class and to notify when the jobs are complete. The Message window echoes all print requests.

The Print_Version procedure uses the Options parameter to allow you to change the printout format or to request multiple copies for your print.

Parameters

`The_Version : Directory.Version;`

Specifies the version of the object to be sent to the print queue.

`Options : String := "<DEFAULT>";`

Specifies the options to be used in formatting output. The following is a list of the options available for use in the Options parameter to format output. Note that the Options parameter uses the special name "<DEFAULT>". When this special name is used, the system looks in the session switch file for the options set in the Queue.Options switch. If the switch file is not accessible, the system uses the options "FORMAT=>(Wrap, System_Header)". These options are no longer true and must be respecified if any of the options are changed by substituting an option for the "<DEFAULT>" special name. One of the following three options must be specified: Original_Raw, Raw, or Format. Unless otherwise specified in the Options parameter, the Boolean options Original_Raw, Raw, and Spool_Each_Item are false. The other options take the defaults specified below.

Banner_Page_User_Text=***string***

Specifies a string (with a maximum of 60 characters) that will be printed on the banner page, beneath the banner and above the system-generated information.

Class=*string*

Specifies the name of the class, where *string* is the class to which the print request is queued. The class determines the device that will handle the print request. Note that the specified class must exist and must be associated with an enable device. If this option is not specified, the class is the default class.

Copies=*positive integer*

Specifies the number of copies to be printed, where *positive integer* is the number of copies to be printed. Copies are generated one at a time, and other jobs may intervene between copies. The default is 1 copy.

Format options

The Format option is an options parameter within the Options parameter that can be used to specify format options Wrap, Truncate, Numbering, System_Header, Width, Length, and Tab_Width. Options must appear in parentheses—for example, `FORMAT=>(Wrap, Width=77)`.

Length=*positive integer*

Specifies the total number of printed lines per page, including headers and footers.

The default length is 60 lines. Note that, by default, the Rational Printer is set to eject a page after 66 lines if a formfeed is not encountered earlier. To print pages longer than 66 lines, you must change the appropriate setting on the Rational Printer (see "Printer Operations and Maintenance" in the *Rational R1000 Development System: System Manager's Guide*) in addition to increasing the Length value.

The number of lines in the body text for each page is automatically adjusted to accommodate any combination of a one-line header, a one-line footer, or a system page header. However, if you specify a multiple-line header or footer, you must decrease the Length value for every additional line beyond the expected one.

Numbering Boolean

Specifies whether to provide line numbering. The default is false.

System_Header Boolean

Specifies whether to print the system page header on each page. The system header is the name of the object and a page number. If there is a user-specified header, the system page header appears above it. The default is false.

Tab Width=*positive integer*

specifies number of spaces with which to replace a Tab character (Ascii.Ht). The default is 8. A value of 0 specifies no replacement.

Truncate Boolean

Specifies whether to truncate lines that are longer than Width. The default is false. If both Truncate and Wrap are set to true, Wrap is assumed to be true.

Width=*positive integer*

Specifies the maximum number of printable characters per line, where *positive integer* is the number of characters.

The default width is 80 columns. Note that the Rational Printer itself can be set to wrap after 80 columns. To print wider pages, you must change the appropriate setting on the Rational Printer (see "Printer Operations and Maintenance" in the *Rational R1000 Development System: System Manager's Guide*) in addition to increasing the value of Width.

The Wrap and Truncate options specify what to do with lines that are longer than Width.

Wrap Boolean

Specifies whether to wrap lines that are longer than Width. The default is false. If both Truncate and Wrap are set to true, Wrap is assumed to be true. The wrapped portions of wrapped lines do not receive a new line number.

Original_Raw Boolean

For use on machines low on space when large files need to be printed. This option prints without using space to make the print spooler copy. The file can be spooled only to a local device. Each file is spooled separately (that is, it ignores the Spool_Each_Item option). A message is sent when printing is complete (that is, it ignores the Notify option). A banner page is printed (that is, it ignores the Banner_Page_User option).

Notify literal

Specifies the manner of notification after a print request is completed. By default, an informative message is sent to the Message window. The available types are None, Message (the default), and Mail (reserved for future development). Remote requests, under normal conditions, will also notify you.

Raw Boolean

Specifies whether the printer should interpret the input. Prints the file without interpreting characters (that is, without recognizing formfeeds or linefeeds). This is useful for preformatted text or binary data. Using this option turns off other options. It does not provide a system or user header.

Spool_Each_Item Boolean

Specifies whether to spool each file indicated by the Name parameter as a separate job. When true, each file has its own banner page. When false, a single banner page is printed. The default is false.

```
Banner : String := "<DEFAULT>";
```

Specifies the string that appears on the single banner page that precedes the printout. The string you supply is truncated at 11 characters. If the null string is specified, a banner page will not be generated. The special name "<DEFAULT>" refers to the banner set in the username's session switch file (or the username if one is not specified).

```
Header : String := "<DEFAULT>";
```

Specifies a line of text that appears at the top of each page of the printout. Any nonnull string (including blank characters) constitutes a user-specified header. A blank line is automatically inserted below the user-specified header to separate the header text from the printout.

If the Options parameter requests a system page header in addition to the user-specified header, then the system header appears first, followed by the user-specified header.

The user-specified header can be longer than the Width option; however, a lengthy header is not wrapped automatically. You must include linefeeds in a header if you want it to wrap onto multiple lines.

The number of lines in the body text for each page is automatically adjusted to accommodate any combination of a one-line header, a one-line footer, or a system page header. However, if you specify a multiple-line header, you must decrease the Length value for every header line beyond the expected one. The special name "<DEFAULT>" refers to the header set in the username's session switch file.

```
Footer : String := "<DEFAULT>";
```

Specifies a line of text that appears at the bottom of each page of the printout. Any nonnull string (including blank characters) constitutes a user-specified footer. A blank line is automatically inserted above the user-specified footer to separate the footer text from the printout.

The user-specified footer can be longer than the Width option; however, a lengthy footer is not wrapped automatically. You must include linefeeds in a footer if you want it to wrap onto multiple lines.

The number of lines in the body text for each page is automatically adjusted to accommodate any combination of a one-line header, a one-line footer, or a system page header. However, if you specify a multiple-line footer, you must decrease the Length value for every footer line beyond the expected one. The special name "<DEFAULT>" refers to the footer set in the username's session switch file.

References

SJM, Session Switches

---

procedure Register

```
procedure Register (Device  : String     := "";
                    Class   : Class_Name := "");
```

Description

Registers the specified device with the print spooler and associates the specified device with the specified class.

The specified device and class must already exist (see the Create and Add procedures).

More than one class can be registered to a single device, and a single class can be registered to more than one device. When one class is registered to multiple devices, print requests submitted to that class are handled by the first available device.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Device :  String := "";
```

Specifies the device by physical line number. The line number takes the format *terminal_n*, where *n* is the number of the port to which the device is attached. The default is no device.

```
Class :  Class_Name := "";
```

Specifies the class to be associated with the device. The default is no class. The class must already exist (see the Create procedure). A remote machine name cannot be used to register a class with a remote machine. Registering must be performed on the same machine as the class to be registered for that machine.

Example

The command:

```
   queue.register  ("terminal_40","lp");
```

registers class LP with the device on physical line 40. If the device on line 40 is the only device associated with class LP, then all requests to class LP are routed to device 40.

References

procedure Add

procedure Create

procedure Unregister

---

procedure Remove

```
procedure Remove (Device    : String  := "";
                  Immediate : Boolean := False);
```

Description

Removes the device from the print spooler.

The value of the Immediate parameter determines whether the procedure waits until the current print request has finished before removing the device. If Immediate is false (the default), the procedure waits for the device to finish processing the current print request. If Immediate is true, the device is removed immediately, and the interrupted print request is requeued.

The Remove procedure effectively reverses all aspects of the Enable and Register procedures. The Remove procedure disables this device, dissociates it from its classes, and then removes the device from the print spooler. (Note that the Unregister procedure dissociates a device from a class but leaves the device known to the print spooler.)

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Device : String := "";
```

Specifies the device by physical line number. The line number takes the form *terminal_n*, where *n* is the number of the port to which the device is attached. The default is no physical line number.

```
Immediate : Boolean := False;
```

Specifies whether the procedure waits until the current print request has finished before removing the device. If false (the default), the procedure waits for the device to finish processing the current print request. If true, the device is removed immediately and the interrupted print request is requeued.

Example 1

The command:

```
queue.remove ("terminal_40",true);
```

removes the device on physical line 40 from the spooler. Because of the parameter true, the device is removed immediately.

Example 2

The command:

```
queue.remove ("terminal_40");
```

removes the device on physical line 40 from the spooler. Because of the default parameter false, the print request currently printing on device 40 completes before the device is removed.

References

procedure Add

procedure Enable

procedure Register

---

procedure Restart_Print_Spooler

```
procedure Restart_Print_Spooler;
```

Description

Starts or restarts the print spooler.

If the spooler is already running, this procedure has no effect. The spooler must be running in order to successfully execute any queue operation.

Execution of this procedure requires that the executing job have operator capability.

procedure Unregister

```
procedure Unregister (Device : String     := "";
                      Class  : Class_Name := "");
```

Description

Dissociates the device from the specified class.

The Unregister procedure reverses the Register procedure. The Unregister procedure dissociates a device from a class but leaves the device known to the print spooler. (Note that the Remove procedure disables the device, dissociates it from its classes, and then removes the device from the print spooler.)

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Device : String := "";`

Specifies the device by physical line number. The line number takes the form *terminal_n*, where *n* is the number of the port to which the device is attached. The default is no physical line number.

`Class : Class_Name := "";`

Specifies a class that is registered to the device. The default is no class. A remote machine name may not be specified to unregister a device on a remote machine.

Example

The command:

```
   queue.unregister ("terminal_40","lp");
```

dissociates class LP from the device on physical line 40.

References

procedure Add

procedure Create

procedure Register

end Queue;

package Scheduler

The medium-term scheduler tracks jobs and regulates their access to CPU, memory, and disk resources. The procedures in package Scheduler allow the user to monitor and fine-tune various aspects of the scheduler's allocation of resources to jobs. By fine-tuning the scheduler, the user can cause it to devote resources to improving the performance of interactive jobs or increasing the throughput of batch jobs.

Typically, a procedure to tailor the scheduler parameters for a system is created and a call to it is inserted in the !Machine.Initialize procedure so that the scheduler parameters are set up each time the system is booted. Sometimes specific jobs or servers will have calls to the scheduler procedure to establish special conditions to optimize their performance.

The operations that set values in this package require that the job executing the procedure have operator capability. Operations that display values do not require any special access.

With the Set procedure, the user can set values for the various scheduler parameters that enable, disable, and adjust CPU scheduling, memory scheduling, disk scheduling, and background job streams. (Each of these topics is discussed in its own section below.) With the Display procedure, the user can display the current values for the scheduler parameters.

The State procedure displays indexes of overall system activity such as run load, disk wait load, withheld task load, and number of available memory pages. The following also return specific information about system activity:

Get_Disk_Wait_Load

Get_Withheld_Task_Load

Get_Run_Queue_Load

Package Scheduler also provides procedures and functions that allow the user to enable, disable, and get information about individual jobs. The Disable and Enable procedures suspend and resume a given job. The State procedure displays information about resource allocation to individual jobs, and the Display procedure displays information specifically about background jobs on the background job streams. Along with the State and Display procedures, the following return specific information about individual jobs:

Display	Get_Cpu_Priority
Get_Cpu_Time_Used	Get_Job_Kind
Get_Job_State	Get_Wsl_Limits
State	

Jobs

A *job* is a set of Ada tasks that act together. A job is initiated each time the user edits an object or executes a command. For scheduling purposes, some jobs are associated with (or mapped to) others. For example, the aggregate of jobs initiated by editing images and objects within a single session are scheduled as a single job.

Job Numbers

Whenever a job starts or is created, it is assigned a unique *Job_Id* (job identification number), which is a number from 0 through 255. The procedures in package Scheduler manipulate jobs by job number, or Job_Id. Therefore, jobs and job numbers are often referred to as Job_Id in the description of individual commands.

Foreground and Background Jobs

Jobs are divided into two major classes for purposes of allocating resources:

- *Foreground jobs* are typically highly interactive and require fast response. They are allocated the majority of system resources, and the scheduler attempts to guarantee that each foreground job makes satisfactory progress.
- *Background jobs* are batch jobs and do not require especially fast response. They are allocated the remaining resources after foreground jobs are handled. The scheduler does not attempt to guarantee that each background job makes progress.

The treatment of foreground and background jobs is described more precisely under "CPU Scheduling," below.

Job Kinds

Within the general division of foreground and background, jobs are allocated resources according to their *kind*. In some cases, a job's kind is determined internally by the Environment; in other cases, a job's kind is determined by the user. The user can use the Get_Job_Kind function to determine a given job's kind.

The kinds of jobs are:

- *Core editor (Ce) jobs* include operations for editing images—for example, commands that control the cursor or search for strings. The Environment determines precisely which operations count as core editor jobs.
- *Object editor (Oe) jobs* include operations for debugging programs and for editing structured objects—for example, commands that select objects. The Environment determines precisely which operations count as object editor jobs.

- *Attached jobs* are commands entered by users that are not core editor or object editor operations. While an attached job executes, the Message window banner displays Running and the terminal is unavailable for other operations.
- *Detached jobs* result either from interrupting an attached job using [Control][G] (the Job.Interrupt procedure) or from entering a command using [Control][Promote] (the Command.Spawn procedure). The terminal can be used for other operations while detached jobs execute.
- *Server jobs* are background jobs that must always have resources available when needed—for example, the print spooler. The user can designate a job as a server using the Set_Job_Attribute procedure.
- *Terminated jobs* are not allocated any resources. A terminated job remains in the system until its job number is reused for another job.

(Note that all job kinds are defined as enumerations of the Job_Kind type.)

Job kinds are related to job classes as follows:

- Core editor and object editor jobs are allocated foreground resources.
- Detached jobs and servers are allocated background resources.
- Attached jobs receive foreground resources until a certain amount of time has elapsed, after which attached jobs receive background resources. (The time limit on attached jobs is determined by the Foreground_Time_Limit scheduler parameter, which the user can set using the Set procedure.) Having a time limit on foreground jobs induces users to run long jobs in the background, rather than depleting foreground resources.

Job States

Jobs typically pass through various *states*: Run, Idle, Wait, Disable, and Queue. These states are enumerations of the Job_State type. The user can use the Get_Job_State function to find out a given job's current state.

Foreground and background jobs alternate between the Run and Wait states as they execute:

- A job in the Run state (also called a *running job*) is either currently consuming CPU time or eligible to consume CPU time. That is, at any given time, there may be several running jobs, of which only one is actually using CPU time while the rest wait their turn.
- A job in the Idle state is not executing. It uses no CPU time and has no unblocked tasks. For example, jobs that are waiting for I/O or that have all tasks are in the Idle state. Also, jobs such as the print spooler are in the Idle state until they are called.
- A job in the Wait state (also called a *withheld job*) is temporarily ineligible for CPU time. The scheduler puts a job in the Wait state if:

  - The job has already used more than its share of CPU time and the system load is too high.
  - The job is waiting for disk resources and the disk wait load is too high.
  - The job is using pages of memory that are needed to replenish the reserve supply of available pages.
- A withheld job returns to the Run state when the scheduler determines that the job is eligible for resources again.
- A job in the Disabled state is not executing. It has been rendered ineligible for CPU time by users or programs executing the !Commands.Job.Disable procedure or the Scheduler.Disable procedure.
- The Queued state is relevant for certain kinds of background jobs—namely, all detached jobs and those attached jobs that have passed the Foreground_Time_Limit. Jobs such as these are queued for resources on one of several background job streams (see "Background Job Streams," below). However, because only a restricted number of jobs on each stream can be in the Run or Wait state at a time, the remaining jobs are put in the Queued state. In other words, a job in the Queued state is a job that is waiting on a background job stream until the executing jobs ahead of it have completed or have been moved to another stream.

Scheduling Review Interval

The scheduler makes scheduling decisions every 100 milliseconds. (These 100-millisecond-intervals are called *review intervals.*) At the end of every review interval, the scheduler reviews the actual usage of resources during the last 100 milliseconds and then, based on the actual usage, decides whether to withhold jobs during the next 100 milliseconds to keep resource usage within certain limits. Note that the scheduler does not actually control the allocation of resources, but rather it monitors and adjusts resource consumption to maintain a balance among different kinds of jobs. The scheduler itself is not subject to scheduling; it uses 0.5–1% of CPU resources.

The following sections give a partial description of the scheduler's effect on CPU scheduling, memory scheduling, and disk scheduling. These sections contain the information the user needs about the scheduler in order to set the scheduler parameters using the Set procedure.

CPU Scheduling

CPU time is distributed among foreground and background jobs, with preference given to foreground jobs. Because foreground jobs need to make consistent progress with the best possible performance, CPU time is made available first to foreground jobs and then to background jobs. However, to prevent foreground jobs from consuming all CPU resources, a minimum percentage of CPU time can be designated for background jobs. This minimum percentage is determined by the Percent_For_Background scheduler parameter, which the user can set using the Set procedure. (Note that even though some CPU time is guaranteed, background jobs will not make progress unless they also have disk and memory resources.)

If there are no background jobs, foreground jobs can receive the CPU time that was reserved for background jobs. Similarly, background jobs can receive more than their reserved CPU time, provided that no foreground jobs need time.

Foreground Jobs

The scheduler follows separate policies for scheduling the CPU time that is allocated to foreground and background jobs. To ensure consistent progress for all foreground jobs, the scheduler attempts to give each session that has foreground jobs an equal share of the foreground CPU time. That is, if two users have foreground jobs, each user's session is given half of the available foreground CPU time. In contrast, individual background jobs are given CPU time according to their Ada task priority and to their placement on a background job stream (see "Background Jobs," below).

Foreground CPU scheduling applies to core editor jobs, object editor jobs, and attached jobs. However, to discourage users from running long jobs in the foreground, the scheduler can be adjusted to give foreground resources to attached jobs only for a limited time. (The time limit is set by the Foreground_Time_Limit scheduler parameter.) After the foreground time limit has expired for an attached job, the job is subject to background CPU scheduling.

To schedule foreground time equitably, the scheduler does the following at the end of each review interval:

1. Determines how much foreground CPU time each job has used during the current review interval.
2. Calculates the ideal CPU usage for each job, giving a fair share to each session that had foreground jobs during the interval.
3. Compares the actual usage to the ideal usage and determines whether the job has used more or less time than it should have. The *foreground budget* for each job is credited or debited accordingly.
4. Decides whether to withhold a job for the next interval. A job is withheld (put in the Wait state) if both of the following are true:
   a. The job has accumulated an overall debt in its foreground budget (that is, the value of the job's foreground budget is negative).
   b. The run load exceeds a preset level.
5. Decides whether to release jobs that were withheld from previous intervals and return them to the Run state. A job is released after it has accumulated enough credit in its foreground budget over one or more review intervals to make up for whatever debt it has previously incurred.

Run Load

The run load is the average number of tasks that require CPU time during a review interval. Tasks are counted if they are currently consuming CPU time or are eligible to consume CPU time. Withheld and idle tasks are not reflected in the run load. The run load is averaged over a review interval and then multiplied by 100 so that it appears as an integer. For example, if an average of 1.3 tasks are in the Run state, the run load is 130.

The scheduler uses the run load to determine whether or not a job can be withheld after that job has used more than its share of CPU time. The user can specify the minimum run load at which the scheduler can withhold jobs by using the Set procedure to set the Withhold_Run_Load parameter.

Number of Withheld Jobs

By default, the scheduler can withhold only one additional job at the end of a given review interval, no matter how many jobs are eligible for withholding after that interval. (However, there is no restriction on the total number of withheld jobs at any given time, because multiple withheld jobs can accumulate after a number of intervals.) The user can permit the scheduler to withhold more than one job per review interval by changing the value for the Withhold_Multiple_Jobs scheduler parameter to true. (See the Set procedure.)

Foreground Budget

As a job uses more or less than its fair share of CPU time, the job's foreground budget is debited or credited accordingly at the end of each review interval. The value of a job's budget at the end of a given interval therefore represents the net debt or credit accumulated over successive intervals. If, on the balance, the job has used more than its allocated time, its budget value is negative. For a withheld job, this negative value expresses how much time the job must accumulate over subsequent intervals in order to be released and returned to the Run state. If, on the other hand, the job has used less than its overall allocated time, its budget value is positive. A positive budget value prevents the job from being withheld and expresses how much extra time the job can use before going into debt. If the job has used exactly as much time as it was allocated, the budget value breaks even at 0.

The scheduler imposes a limit on the amount of accumulated credit or debt a job can have. That is, no matter how much extra time a job has used, there is a maximum overall debt that the job can incur. Consequently, if the job is withheld, there is a limit to the amount of credit it has to accumulate before it can run again. Similarly, no matter how little time the job used relative to its allotment, there is a maximum overall credit that the job can earn. Consequently, there is a limit to how much extra time the job can use before going into debt.

The user can adjust the credit and debit limits on the foreground budget by using the Set procedure to set the Max_Foreground_Budget and Min_Foreground_Budget scheduler parameters, respectively. The wider the range between the Max_Foreground_Budget and Min_Foreground_Budget values, the more sensitive the scheduler is to giving jobs equal time. When the range is narrower, the distribution of CPU time is less equal.

Background Jobs

As a group, background jobs (detached jobs, servers, and aged attached jobs) are guaranteed a percentage of CPU time, as determined by the Percent_For_Background scheduler parameter. However, the scheduler does not track the amount of time used by each background job, nor does the scheduler attempt to ensure that each job is allotted a fair share of the available CPU time. Instead, the allocation of CPU time is determined by Ada task priorities.

Background Job Streams

Without a guarantee of equal CPU time, it is possible for a single long-running background job to block a number of shorter jobs. To avoid this, ***background job streams*** can be set up to queue long-running jobs to expedite shorter jobs. Only two kinds of background jobs are subject to queuing on the background job streams—namely, detached jobs and attached jobs that have run longer than the Foreground_Time_Limit. Servers are not subject to queuing on these streams.

By default, there is one background job stream, although the user can arbitrarily set up streams by using the Set procedure to set the Background_Streams scheduler parameter. The Display procedure displays information about each background job stream.

Job Stream Time Limits

Each background job stream has an associated time limit, which specifies the maximum amount of elapsed time a job can run on that stream. If a job that is running on a stream has not yet finished when the time limit is reached, the job is queued onto the next stream.

For example, the user could set up three streams with the following time limits:

Stream 1	2 minutes
Stream 2	5 minutes
Stream 3	20 minutes

With these limits, a job queued on stream 1 can run for 2 minutes. If the job has not finished within that time, it is queued onto stream 2, where it waits its turn to run. (Meanwhile, another job on stream 1 can now run.) Once the job is eligible to run on stream 2, it can run for 5 more minutes. If the job requires more than 5 minutes, it is queued onto stream 3, so that subsequent jobs on stream 2 can run. Once the job is eligible to run on stream 3, it can run for another 20 minutes. If the job requires even more time, it is moved to the bottom of the queue on stream 3. After the jobs ahead of it have finished or have been requeued, the job gets another 20 minutes, and so on.

The time limits for each stream are determined by the Stream_Time parameters (see the Set procedure).

Note that a job on a job stream uses temporary disk space that is not reclaimed until the job is done. Allowing many jobs to accumulate on multiple job streams can cause a shortage of disk space. If the Stream_Time value for a job stream is low, jobs on that stream are more likely to be requeued before they can finish. A high Stream_Time value permits jobs to complete without being requeued.

Number of Runnable Jobs on a Stream

Each stream has an associated prescribed number of jobs that can be running at a given time. These numbers are specified by the Stream_Jobs parameters (see the Set procedure). If a job stream contains more jobs than are permitted to run at a given time, the excess jobs are put in the Queued state to wait until the jobs ahead of them are finished or requeued to the next stream.

For example, if the Stream_Jobs value for a stream is 2, then only two jobs on that stream can be in the Run or Wait state at a time. Therefore, if ten jobs are on that stream, eight jobs must be in the Queued state.

Strict Stream Policy

Although the Stream_Jobs parameters specify a prescribed number of runnable jobs per stream, the actual number of running jobs on each stream is also determined by the presence or absence of ***strict stream policy***. When strict stream policy is in effect, the Stream_Jobs value for a given stream is always the maximum number of jobs that can run concurrently on that stream. In contrast, when strict stream policy is not in effect, the number of jobs on a given stream can exceed the relevant Stream_Jobs value, provided that other streams are empty. However, although the distribution of runnable jobs across streams is affected, the total number of jobs running on all streams taken together cannot exceed the total of the Stream_Jobs values for all the streams.

For example, under strict stream policy, a system with three streams might have the following Stream_Jobs values:

Stream 1	2 jobs
Stream 2	1 jobs
Stream 3	1 jobs

If jobs are queued in all three streams, a maximum of four jobs can be running—specifically, only the first two jobs in stream 1, the first job in stream 2, and the first job in stream 3. If streams 1 and 3 are empty, the maximum number of running jobs across all streams is only one, because stream 2 has a value of 1. Strict stream policy prohibits extra jobs from running on stream 2, no matter how many jobs are queued.

If strict stream policy is not in effect, then the maximum number of running jobs is always four, even when some streams are empty. That is, if streams 1 and 3 are empty, up to four jobs can run on stream 2, because the empty streams contribute their Stream_Jobs values to the nonempty stream.

The following Stream_Jobs values make sense only if strict stream policy is not in effect:

Stream 1	3 jobs
Stream 2	0 jobs
Stream 3	0 jobs

If strict stream policy were in effect, jobs queued on streams 2 and 3 would never run, because their Stream_Jobs values are 0. However, because strict stream policy is not, by default, in effect, jobs queued on streams 2 or 3 can run whenever stream 1 has fewer than three jobs in its queue.

Strict stream policy is controlled by the value of the Strict_Stream_Policy scheduler parameter.

Memory Scheduling

Each job uses pages of main memory while executing. On most R1000 systems, the memory size is 32,768 pages, each of which contains 1,024 bytes. The number of pages in memory is defined by the !Lrm.System.Memory_Size constant.

The scheduler dynamically adjusts the allocation of memory to give pages to jobs that need more and to reclaim pages from jobs that need fewer. The number of pages used by a job is called the job's *job working set size.* To prevent any one job from consuming a disproportionate amount of memory resources, the scheduler places a limit on each job's working set size. This limit, called the *job working set limit*, is the maximum number of pages a job can use without penalty.

Jobs started by the Environment or by the system daemon have fixed working set limits. The user can specify these working set limits by setting the Environment_Wsl and Daemon_Wsl scheduler parameters, respectively. In contrast, the working set limit for each user job is determined dynamically. When a job is created, it is given an initial working set limit, which is adjusted at regular intervals to ensure adequate allocation of pages to all jobs.

The value of a job's initial working set limit depends on what kind of job it is (see "Job Kinds," above). The values of scheduler parameters such as Min_Ce_Wsl, Min_Oe_Wsl, and so on determine the initial working set limit for each kind of job.

At the end of every review interval, the scheduler checks each job's working set size. If the job's working set size exceeds its working set limit, the scheduler increases the job's limit by a fixed number of pages. The user can specify this number by setting the Wsl_Growth_Factor scheduler parameter.

The scheduler tries to keep each job's working set limit close to its working set size. Therefore, in addition to automatically increasing the working set limit ten times a second, the scheduler automatically decreases each job's working set limit every 5 seconds. The limit is decreased by a fixed number of pages, which the user can specify by setting the Wsl_Decay_Factor scheduler parameter.

The growth and decay of the working set limits for each kind of job are kept within a range of values that are specified by scheduler parameters. For example, the lowest possible working set limit for core editor jobs is determined by the Min_Ce_Wsl parameter (which is also the value of the initial working set limit). The highest possible working set limit for core editor jobs is determined by Max_Ce_Wsl. Similarly, the range for object editor working set limits is determined by Min_Oe_Wsl and Max_Oe_Wsl, and so on for attached jobs, detached jobs, and servers. These parameters can be used to give preference to some kinds of jobs over others. For example, the default values for Max_Detached_Wsl and Max_Attached_Wsl give background user jobs more than twice as much memory as foreground user jobs. The user can temporarily override the maximum and minimum working set limits for a given job by using the Set_Wsl_Limits procedure.

The scheduler reserves a number of pages for distribution among jobs that need more memory. If the number of available pages falls below a given limit, the scheduler withholds jobs as needed and contributes the freed pages to the reserve. The user can specify the minimum number of pages kept on reserve by setting the Minimum-_Available_Memory scheduler parameter.

Page Withdrawal

During every review interval, the scheduler withdraws a fixed number of pages from memory. Withdrawn pages are earmarked for possible removal from the jobs that are using them. However, a withdrawn page is not actually taken away from a job unless that job exceeds its working set limit during the review interval.

Withdrawing pages serves two purposes, namely:

- To help account for pages that are shared by multiple jobs. When a shared page is withdrawn, its use is charged to the first job to request it again. Accurately accounting for the use of shared pages is necessary for determining the working set size of each job.
- To earmark pages from overallocated jobs for potential use by underallocated jobs. If a job exceeds its working set limit, the excess pages are withdrawn so that they can be allocated to other jobs.

The user can specify the number of pages withdrawn per review interval by setting the Page_Withdrawal_Rate scheduler parameter.

Disk Scheduling

The scheduler measures disk activity through an index called the ***disk wait load***. The disk wait load is the average number of tasks waiting on disk operations, including page faults and disk I/O operations. The disk wait load is averaged over an internally determined interval of time and then multiplied by 100, so that it is expressed as an integer. For example, if an average of 1.5 tasks are waiting for pages from disk at a given time, the disk wait load is 150.

The scheduler regulates disk activity by monitoring the disk wait load and withholding one or more jobs when the load exceeds a certain limit. The user can set this limit by setting the Max_Disk_Load scheduler parameter. The user can also ensure a minimum level of disk activity by setting the Min_Disk_Load scheduler parameter. Together, the Max_Disk_Load and Min_Disk_Load parameters define a range of acceptable stress on the disks. The wider the range, the less sensitive the scheduler is to changes in the disk wait load.

subtype Cpu_Priority

```
subtype Cpu_Priority is Natural range 0 .. 6;
```

Description

Identifies priority of access to CPU resources.

A Cpu_Priority of 0 is the lowest; a Cpu_Priority of 6 is the highest. The higher a job's priority, the more CPU time the job gets. Background jobs have a Cpu_Priority of 0; foreground jobs have a Cpu_Priority of 6.

procedure Disable

```
procedure Disable (Job : Job_Id);
```

Description

Suspends temporarily the job with the specified Job_Id.

A disabled job can be resumed with the Enable procedure.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

References

procedure Enable

function Disk_Waits

```
function Disk_Waits (Job : Job_Id) return Long_Integer;
```

Description

Returns the number of disk waits the specified job has had since last initialized.

A disk wait occurs whenever a job has to wait for disk resources. The number of disk waits is derived from a combination of page faults and disk I/O operations. A high number of disk waits indicates heavy disk activity; a low number indicates light disk activity.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Long_Integer;
```

Specifies the number of disk waits.

procedure Display

```
procedure Display (Show_Parameters  : Boolean := True;
                   Show_Queues      : Boolean := True);
```

Description

Displays the current values for the scheduler parameters along with information about background job streams in the current output window.

For each background job stream, this procedure displays the value of the Stream_Time parameter, a list of the jobs currently in the stream, and the number of minutes each job has been in the stream. An asterisk next to a job indicates that the job is currently running—that is, it has Job_State Run. A job without an asterisk has Job_State Queued.

The user can change the values for the scheduler parameters by using the Set procedure.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Show_Parameters : Boolean := True;`

Requests, when true, a display of the current values for the scheduler parameters.

`Show_Queues : Boolean := True;`

Requests, when true, a display of background job stream information.

Example 1

The command:

```
scheduler.display (true,false);
```

produces a display such as the following:

```
Cpu_Scheduling     : ENABLED
Disk_Scheduling    : ENABLED
Memory_Scheduling  : ENABLED

Percent_For_Background           : 20%
Min_ and Max_Foreground_Budget   :-250 .. 250 milliseconds
Withhold_Run_Load                : 130
Withhold_Multiple_Jobs           : FALSE

Environment_Wsl            : 11000 pages
Daemon_Wsl                 : 200 pages
Min_ and Max_Ce_Wsl        : 400 .. 1000 pages
Min_ and Max_Oe_Wsl        : 250 .. 2000 pages
Min_ and Max_Attached_Wsl  : 50 .. 4000 pages
Min_ and Max_Detached_Wsl  : 50 .. 4000 pages
Min_ and Max_Server_Wsl    : 400 .. 1000 pages
Min_Available_Memory       : 1024 pages
Wsl_Decay_Factor           : 50 pages/5 seconds
Wsl_Growth_Factor          : 50 pages/100 milliseconds
Page_Withdrawal_Rate       : 1*640 pages/second

Min_ and Max_Disk_Load : 200 .. 250

Foreground_Time_Limit     : 1800 seconds
Background_Streams        : 3
Strict_Stream_Policy      : FALSE
Stream_Time and _Jobs 1 : 2 minutes, 3 jobs
Stream_Time and _Jobs 2 : 58 minutes, 0 jobs
Stream_Time and _Jobs 3 : 0 minutes, 0 jobs
```

Example 2

The command:

```
   scheduler.display (false,true);
```

produces a display such as the following in the current output window:

```
   Stream 1           2:00
      252   9:14
    * 223   8:00
      253   3:46
    * 219   3:29
      238   0:48
   Stream 2           58:00
    * 220   5:00
      261   2:49
   Stream 3           0:00
    * 222   2:00
      254   1:26
```

This display shows five jobs in stream 1, of which two are currently running and the rest are queued. The job that has been in that stream the longest is listed first and has been there for 9 minutes and 14 seconds. It also shows jobs in streams 2 and 3, using the same format as used in stream 1.

References

procedure Set

procedure Enable

```
procedure Enable (Job : Job_Id);
```

Description

Resumes the execution of the specified disabled job.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

References

procedure Disable

function Enabled

```
function Enabled (Job : Job_Id) return Boolean;
```

Description

Returns a Boolean indicating whether the specified job is currently enabled for execution.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Boolean;
```

Specifies, when true, that the job is enabled.

function Get

```
function Get (Parameter : String) return Integer;
```

Description

Returns the current value for the specified scheduler parameter.

The names and possible values of the scheduler parameters are listed under the Set procedure.

Parameters

```
Parameter : String;
```

Specifies any of the strings listed under the Set procedure.

```
return Integer;
```

Specifies the current value for the specified scheduler parameter.

References

procedure Set

function Get_Cpu_Priority

```
function Get_Cpu_Priority (Job : Job_Id) return Cpu_Priority;
```

Description

Returns the priority at which the specified job is currently running.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Cpu_Priority;
```

Specifies the value range, from 0 (lowest) through 6 (highest). The higher the priority, the more CPU time the job gets.

function Get_Cpu_Time_Used

```
function Get_Cpu_Time_Used (Job : Job_Id) return Milliseconds;
```

Description

Returns the number of milliseconds of CPU time used by the specified job since that job began.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Milliseconds;
```

Specifies the amount of CPU time in milliseconds.

procedure Get_Disk_Wait_Load

```
procedure Get_Disk_Wait_Load (Last_Sample     : out Load_Factor;
                              Last_Minute     : out Load_Factor;
                              Last_5_Minutes  : out Load_Factor;
                              Last_15_Minutes : out Load_Factor);
```

Description

Returns the average number of tasks waiting for pages from disk.

The number is averaged over each of four sampling intervals: 100 milliseconds, 1 minute, 5 minutes, and 15 minutes. Each average is multiplied by 100, so that it appears as an integer.

Parameters

`Last_Sample : out Load_Factor;`

Specifies disk wait load averaged over the last 100 milliseconds.

`Last_Minute : out Load_Factor;`

Specifies disk wait load averaged over the last minute.

`Last_5_Minutes : out Load_Factor;`

Specifies disk wait load averaged over the last 5 minutes.

`Last_15_Minutes : out Load_Factor;`

Specifies disk wait load averaged over the last 15 minutes.

function Get_Job_Attribute

```
function Get_Job_Attribute (Job       : Job_Id;
                            Attribute : String  := "Kind") return String;
```

Description

Returns the attributes for a job as described in the Set_Job_Attribute procedure.

Parameters

`Job :  Job_Id;`

Specifies the job number.

`Attribute :  String := "Kind";`

Specifies the attribute to be checked. The only attribute currently supported is "Kind", which returns an image of an enumeration of the Job_Kind type.

`return String;`

Returns a string representing an enumeration of the Job_Kind type.

Example

The following program segment illustrates the use of the Get_Job_Attribute function to display the job attribute for a user job number 244:

```
   . . .
      A: Scheduler.Job_Id := 244;
   begin
      Io.Put(Scheduler.Get_Job_Attribute(Job=>A,Attribute=>"Kind"));
   end;
```

References

procedure Set_Job_Attribute

function Get_Job_Descriptor

```
function Get_Job_Descriptor (Job : Job_Id) return Job_Descriptor;
```

Description

Returns the current value of a job's statistics.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Job_Descriptor;
```

Returns a record of the Job_Descriptor type. See the Job_Descriptor type for further information about the contents of that record.

function Get_Job_Kind

```
function Get_Job_Kind (Job : Job_Id) return Job_Kind;
```

Description

Returns the current Job_Kind type of the specified job.

This function specifies one of the six kinds of jobs defined by the Job_Kind type: Ce (core editor), Oe (object editor), Attached, Detached, Server, or Terminated. The scheduler follows a different scheduling policy for each of these job kinds.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Job_Kind;
```

Returns one of the six kinds of job defined by the Job_Kind type: Ce (core editor), Oe (object editor), Attached, Detached, Server, or Terminated. The scheduler follows a different scheduling policy for each of these job kinds.

function Get_Job_State

```
function Get_Job_State (Job : Job_Id) return Job_State;
```

Description

Returns the current Job_State type of the specified job.

Parameters

```
Job : Job_Id;
```

Specifies the job number.

```
return Job_State;
```

Returns one of the five job states defined by the Job_State type: Run, Wait, Idle, Disabled, or Queued. Job_State reflects how the scheduler has disposed of a job—that is, whether the job is earmarked for running or is considered unrunnable for some reason.

procedure Get_Run_Queue_Load

```
procedure Get_Run_Queue_Load (Last_Sample     : out Load_Factor;
                              Last_Minute     : out Load_Factor;
                              Last_5_Minutes  : out Load_Factor;
                              Last_15_Minutes : out Load_Factor);
```

Description

Returns the number of runnable, unblocked tasks, averaged over each of four sampling intervals: 100 milliseconds, 1 minute, 5 minutes, and 15 minutes.

Runnable tasks are neither withheld nor idle, but they are either currently consuming CPU time or eligible to consume CPU time.

Each average is multiplied by 100, so that it appears as an integer.

Parameters

`Last_Sample : out Load_Factor;`

Specifies the run queue load averaged over the last 100 milliseconds.

`Last_Minute : out Load_Factor;`

Specifies the run queue load averaged over the last minute.

`Last_5_Minutes : out Load_Factor;`

Specifies the run queue load averaged over the last 5 minutes.

`Last_15_Minutes : out Load_Factor;`

Specifies the run queue load averaged over the last 15 minutes.

References

SJM, procedure What.Load

procedure Get_Withheld_Task_Load

```
procedure Get_Withheld_Task_Load (Last_Sample     : out Load_Factor;
                                  Last_Minute     : out Load_Factor;
                                  Last_5_Minutes  : out Load_Factor;
                                  Last_15_Minutes : out Load_Factor);
```

Description

Returns the number of tasks that are withheld from running, averaged over each of four sampling intervals: 100 milliseconds, 1 minute, 5 minutes, and 15 minutes.

Each average is multiplied by 100, so that it appears as an integer.

A task (job) is withheld from running if it is consuming more than its share of resources or if it has been queued or disabled.

Parameters

`Last_Sample : out Load_Factor;`

Specifies the number of withheld tasks averaged over the last 100 milliseconds.

`Last_Minute : out Load_Factor;`

Specifies the number of withheld tasks averaged over the last minute.

`Last_5_Minutes : out Load_Factor;`

Specifies the number of withheld tasks averaged over the last 5 minutes.

`Last_15_Minutes : out Load_Factor;`

Specifies the number of withheld tasks averaged over the last 15 minutes.

procedure Get_Wsl_Limits

```
procedure Get_Wsl_Limits (Job      :     Job_Id;
                          Min, Max : out Natural);
```

Description

Returns the minimum and maximum working set limits that are currently in effect for the specified job.

These limits may be the temporary limits set by the Set_Wsl_Limits procedure, or they may be the limits defined by the relevant scheduler parameters (Min_ and Max_Ce_Wsl, Min_ and Max_Oe_Wsl, and the like), which are described under the Set procedure and in the introduction to this package.

Parameters

`Job : Job_Id;`

Specifies the number of the job whose minimum and maximum working set limits the user wants to see.

`Min : out Natural;`

Returns the current minimum working set limit for the given job.

`Max : out Natural;`

Returns the current maximum working set limit for the given job.

References

procedure Set_Wsl_Limits

procedure Use_Default_Wsl_Limits

type Job_Descriptor

```
type Job_Descriptor is
    record
        The_Cpu_Priority              : Cpu_Priority;
        The_State                     : Job_State;
        The_Disk_Waits                : Long_Integer;
        The_Time_Consumed             : Milliseconds;
        The_Working_Set_Size          : Natural;
        The_Working_Set_Limit         : Natural;
        The Milliseconds_Per_Second   : Natural;
        The_Disk_Waits_Per_Second     : Natural;
        The_Maps_To                   : Job_Id;
        The_Kind                      : Job_Kind;
        The_Made_Runnable             : Long_Integer;
        The_Total_Runnable            : Long_Integer;
        The_Made_Idle                 : Long_Integer;
        The_Made_Wait                 : Long_Integer;
        The_Wait_Disk_Total           : Long_Integer;
        The_Wait_Memory_Total         : Long_Integer;
        The_Wait_Cpu_Total            : Long_Integer;
        The_Min_Working_Set_Limit     : Long Integer;
        The_Max_Working_Set_Limit     : Long Integer;
    end record;
```

Description

Specifies a record that contains information returned by many of the functions in this package.

This is a convenient way of storing all of the available information for a particular job.

The_Cpu_Priority component

Contains the priority at which the specified job is currently running, as returned by the Get_Cpu_Priority function. Specifies the value range, from 0 (lowest) through 6 (highest). The higher the priority, the more CPU time the job gets.

The_State component

Contains the current Job_State of the specified job, as returned by the Get_Job_State function. Specifies one of the five job states defined by the Job_State type: Run, Wait, Idle, Disabled, or Queued. Job_State reflects how the scheduler has disposed of the job—that is, whether the job is earmarked for running or is considered unrunnable for some reason.

The_Disk_Waits component

Contains the number of disk waits the specified job has had since last initialized, as returned by the Disk_Waits function. A disk wait occurs whenever the job has to wait for disk resources. The number of disk waits is derived from a combination of page faults and disk I/O operations. A high number of disk waits indicates heavy disk activity; a low number indicates light disk activity.

The_Time_Consumed component

Returns the number of milliseconds of CPU time used by the specified job since that job began.

The_Working_Set_Size component

Contains the number of pages of memory used by the specified job, as returned by the Working_Set_Size function.

The_Working_Set_Limit component

Contains the limit that the scheduler places on the job's working set size. This limit is the maximum number of pages the job can use without penalty. The working set limit for the user job is determined dynamically. When the job is created, it is given an initial working set limit, which is adjusted at regular intervals to ensure adequate allocation of pages to all jobs.

The value of the job's initial working set limit depends on the kind of job it is. The values of scheduler parameters such as Min_Ce_Wsl, Min_Oe_Wsl, and so on determine the initial working set limit for the job.

The Milliseconds_Per_Second component

Contains the number of milliseconds of CPU time the job has used in the last 5-second evaluation interval, as shown in the CPU MS/S field of the display resulting from execution of the State procedure.

The_Disk_Waits_Per_Second component

Contains the number of disk waits the job had in the last 5-second evaluation interval, as shown by the DISK DW/S field of the display resulting from execution of the State procedure.

The_Maps_To component

Contains the core editor (if any) the job is grouped with for CPU scheduling purposes, as shown by the MAP TO field of the display resulting from execution of the State procedure.

The_Kind component

Contains the policy used by the scheduler to allocate resources to the job, as in the Job_Kind type.

In general, Ce, Oe, and Attached jobs are interactive and require enough resources for constant forward progress. Detached jobs (and aged Attached jobs) are not interactive and therefore do not require a constant supply of resources.

The_Made_Runnable component

Contains the number of times the job entered the run state. For further information, see "Job Kinds" in the introduction to this package.

The_Total_Runnable component

Contains the number of times the job was available to enter the run state. For further information, see "Job Kinds" in the introduction to this package.

The_Made_Idle component

Contains the number of times the job entered the idle state. For further information, see "Job Kinds" in the introduction to this package.

The_Made_Wait component

Contains the number of times the job entered the wait state. For further information, see "Job Kinds" in the introduction to this package.

The following three components give information about why the job went into the Wait state.

The_Wait_Disk_Total component

Contains the number of times the job entered the wait state because it was waiting for disk space.

The_Wait_Memory_Total component

Contains the number of times the job entered the wait state because it was waiting for memory.

The_Wait_Cpu_Total component

Contains the number of times the job entered the wait state because it was waiting for CPU time.

The_Min_Working_Set_Limit component

Contains the minimum working set limit that is currently in effect for the specified job.

This limit may be the temporary limit set by the Set_Wsl_Limits procedure, or it may be the limit defined by the relevant scheduler parameters (Min_Ce_Wsl, Min_Oe_Wsl, and the like), which are described under the Set procedure and in the introduction to this package.

The_Max_Working_Set_Limit component

Returns the maximum working set limits that are currently in effect for the specified job.

This limit may be the temporary limit set by the Set_Wsl_Limits procedure, or it may be the limit defined by the relevant scheduler parameters (Max_Ce_Wsl, Max_Oe_Wsl, and the like), which are described under the Set procedure and in the introduction to this package.

subtype Job_Id

```
subtype Job_Id is Machine.Job_Id;
```

Description

Specifies the form of Job_Id.

Job_Ids are assigned uniquely for each instance of system activity. Job_Ids are often referred to as job numbers.

type Job_Kind

```
type Job_Kind is (Ce, Oe, Attached, Detached, Server, Terminated);
```

Description

Determines the policy used by the scheduler to allocate resources to a job.

In general, Ce, Oe, and Attached jobs are interactive and require enough resources for constant forward progress. Detached jobs (and aged Attached jobs; see "Attached" below) are not interactive and therefore do not require a constant supply of resources.

Enumerations

`Attached`

Specifies that the job is a foreground job; the Message window banner displays `Running` while an Attached job executes. An Attached job is scheduled to receive its share of the foreground CPU time until the Foreground_Time_Limit is reached. (Foreground_Time_Limit is set using the Set procedure.) After the time limit is reached, the job is aged, which means that, although it is still attached, it receives a smaller amount of CPU time and is subject to queuing in the background job streams, as if it were detached.

`Ce`

Specifies that the job (a core editor) is scheduled to receive its share of foreground CPU time.

`Detached`

Specifies that the job is running in the background, either started in the background by the !Commands.Command.Spawn procedure or put there by the !Commands.Job.Interrupt procedure. A Detached job is eligible to receive CPU time, although it generally receives less CPU time than an Attached job. A Detached job is subject to queuing on a background job stream.

`Oe`

Specifies that the job (an object editor) is scheduled to receive its share of foreground CPU time.

Server

Specifies that the job is a Server, which is a background job that must always have resources available to it when it needs them. An example of a Server is the print spooler. A Server is allocated the resources of a background job; however, unlike other background jobs, a Server is not subject to queuing in the background job streams, so that it is always eligible to run.

Terminated

Specifies that the job has completed. A Terminated job remains until its number (Job_Id) is reused for another job.

type Job_State

```
type Job_State is (Run, Wait, Idle, Disabled, Queued);
```

Description

Indicates how the scheduler has disposed of a job—that is, whether or not the job is eligible for CPU time.

This information is displayed in the S column of the !Commands.What.Users display.

Enumerations

`Disabled`

Specifies that the job is ineligible for CPU time because an external agent has disabled it, for example, using the !Commands.Job.Disable procedure or the Scheduler.Disable procedure.

`Idle`

Specifies that the job uses no CPU time and has no unblocked tasks. For example, the print spooler remains Idle until called.

`Queued`

Specifies that the job is in a background job stream but not currently in the Run or Wait (withheld) state. A Queued job is a background job that is ineligible for CPU time until one or more other jobs have completed.

`Run`

Specifies that the job is eligible for CPU time, providing its priority is high enough.

`Wait`

Specifies that the job has been withheld from running by the scheduler. (A withheld job is temporarily ineligible for CPU time.) The scheduler puts a job in the Wait state if:

- The job has already used more than its share of CPU time and the system load is too high.
- The job is waiting for pages from disk and the disk wait load is too high.

subtype Load_Factor

```
subtype Load_Factor is Natural;
```

Description

Defines a representation for the load on the system.

When multiplied by 100, the load factor is the number of tasks waiting in a queue. For example, for a given interval, if an average of 1.3 tasks are waiting for a page from disk, then the disk wait load is 130.

subtype Milliseconds

```
subtype Milliseconds is Long_Integer;
```

Description

Defines a way of representing an amount of time in milliseconds.

procedure Set

```
procedure Set (Parameter : String    := "";
               Value     : Integer);
```

Description

Sets the specified scheduler parameter to the specified value.

The Set procedure allows the user to set one parameter at a time. The executing job must have operator capability.

The user can change scheduler parameters to adjust aspects of CPU scheduling, memory scheduling, disk scheduling, and background job streams. The descriptions of the scheduler parameters below assume that the user has read the introduction to this package for general information about the scheduler.

The Display procedure displays the current values for scheduler parameters. The user can also use the Get function to get the current value for a given scheduler parameter.

Parameters

```
Parameter : String := "";
```

Specifies the scheduler parameter whose value is to be set. The acceptable string names for these parameters are listed in the following table.

```
Value : Integer;
```

Specifies integer values for scheduler parameters. The acceptable values are listed in the following table.

There are scheduler parameters for adjusting CPU scheduling, memory scheduling, disk scheduling, and background job streams. The following table summarizes the string names of these parameters, their possible values, and their default values. The parameters are discussed in greater detail following the table.

Scheduler Parameters

Parameter	*Range of Values*	*Default Value*	*Description*
CPU Scheduling			
Cpu_Scheduling	0 or 1 (off or on)	1	Specifies whether to disable (0) or enable (1) CPU scheduling.
Percent_For_Background	0 .. 90 (percent)	20	Specifies the minimum percentage of CPU allocated to background jobs.
Min_Foreground_Budget	-5000 .. 0 (milliseconds)	-250	Specifies the maximum debt that a job's foreground budget can show.
Max_Foreground_Budget	0 .. 5000 (milliseconds)	250	Specifies the maximum credit that a job's foreground budget can show.
Withhold_Run_Load	0 .. 900 (Load_Factor)	130	Specifies the minimum run load at which the scheduler can withhold jobs.
Withhold_Multiple_Jobs	0 or 1 (false or true)	0	Specifies whether or not multiple additional jobs can be withheld at a time.
Memory Scheduling			
Memory_Scheduling	0 or 1 (off or on)	1	Specifies whether to disable (0) or enable (1) memory scheduling.
Environment_Wsl	1 .. memory size (pages)	11000	Specifies the working set limit for the Environment.
Daemon_Wsl	1 .. memory size (pages)	200	Specifies the working set limit for jobs started by the system daemon.
Min_Ce_Wsl	1 .. max_ce_wsl	400	Specifies the minimum working set limit for core editor jobs.
Max_Ce_Wsl	min .. memory size	1000	Specifies the maximum working set limit for core editor jobs.
Min_Oe_Wsl	1 .. max_oe_wsl	250	Specifies the minimum working set limit for object editor jobs.
Max_Oe_Wsl	min .. memory size	2000	Specifies the maximum working set limit for object editor jobs.
Min_Attached_Wsl	1 .. max_attached_wsl	50	Specifies the minimum working set limit for attached jobs.
Max_Attached_Wsl	min .. memory size	4000	Specifies the maximum working set limit for attached jobs.
Min_Detached_Wsl	1 .. max_detached_wsl	50	Specifies the minimum working set limit for detached jobs.
Max_Detached_Wsl	min .. memory size	4000	Specifies the maximum working set limit for detached jobs.
Min_Server_Wsl	1 .. max_server_wsl	400	Specifies the minimum working set limit for servers.
Max_Server_Wsl	min .. memory size	1000	Specifies the maximum working set limit for servers.
Min_Available_Memory	0 .. memory size (pages)	1024	Specifies the minimum number of pages of memory that should always be available for distribution.

Scheduler Parameters *Continued*

Parameter	*Range of Values*	*Default Value*	*Description*
Memory Scheduling *(Continued)*			
Wsl_Decay_Factor	0 .. 2000 (pages)	50	Specifies the amount by which a user job's working set limit is decreased every 5 seconds.
Wsl_Growth_Factor	0 .. 2000 (pages)	50	Specifies the amount by which a user job's working set limit is increased.
Page_Withdrawal_Rate	0 .. 64 (640 pages/second)	1	Specifies the rate at which pages are withdrawn from memory.
Disk Scheduling			
Disk_Scheduling	0 or 1 (off or on)	1	Specifies whether to disable (0) or enable (1) disk scheduling.
Max_Disk_Load	positive (Load_Factor)	250	Specifies the maximum acceptable disk wait load before jobs are withheld.
Min_Disk_Load	positive (Load_Factor)	200	Specifies the minimum acceptable disk wait load.
Background Job Streams			
Foreground_Time_Limit	positive (seconds)	1800	Specifies how long an attached job can run before it gets allocated background resources.
Background_Streams	positive (streams)	3	Specifies the number of background job streams.
Stream_Time N	1 .. 43200 (minutes)	2, 58, 00	Specifies the time limit associated with the job stream numbered N. (Each stream has its own default.)
Stream_Jobs N	0 .. 5 (jobs)	3, 0, 0	Specifies the number of jobs that can be running in the jobs stream numbered N. (Each stream has its own default.)
Strict_Stream_Policy	0 or 1 (false or true)	0	Specifies whether strict stream policy is in effect (true) or not (false).

Parameters for CPU Scheduling

Cpu_Scheduling

Specifies whether to enable or disable CPU scheduling independently of memory or disk scheduling. When CPU scheduling is disabled, all jobs are run according to Ada task priorities; no attempt is made to guarantee equal time to each foreground job. The value of Cpu_Scheduling is either 0 (disabled) or 1 (enabled). The default value is 1 (enabled).

Percent_For_Background

Specifies the minimum percentage of the CPU that is allocated to background jobs. The default value, 20, means that at least 20% of the CPU can be used by the background job stream at any time.

Min_Foreground_Budget

Specifies the limit on how much accumulated debt a job's foreground budget can show. Excess debt is ignored.

The value of Min_Foreground_Budget is a negative number of milliseconds from −5,000 through 0. The default value is −250 milliseconds.

Max_Foreground_Budget

Specifies the limit on how much accumulated credit a job's foreground budget can show. Excess credit is ignored. The wider the range between Max_ and Min_Foreground_Budget, the more sensitive the scheduler is to giving jobs equal time.

The value of Max_Foreground_Budget is a positive number of milliseconds from 0 through 5,000. The default value is 250 milliseconds.

Withhold_Run_Load

Specifies the minimum run load at which the scheduler is permitted to withhold jobs. The higher the value, the more heavily loaded the system must be before the scheduler can withhold a job for exceeding its allocated time.

The value for Withhold_Run_Load must be an integer from 0 through 900. The default value for Withhold_Run_Load is 130. The run load value is the average number of tasks that are eligible to run, multiplied by 100.

Withhold_Multiple_Jobs

Specifies whether the scheduler is restricted to withholding only one more job per review interval (in addition to any jobs that were withheld on previous intervals). When the value for Withhold_Multiple_Jobs is 1 (true), the scheduler can withhold multiple jobs in response to oversubscribed CPU resources. When the value is 0 (false), the scheduler can withhold at most one job at the end of a single interval.

The default value for Withhold_Multiple_Jobs is 0 (false).

Parameters for Memory Scheduling

`Memory_Scheduling`

Specifies whether to enable or disable memory scheduling independently of CPU or disk scheduling. The value of Memory_Scheduling is either 0 (disabled) or 1 (enabled). The default value is 1 (enabled).

`Environment_Wsl`

Specifies the working set limit for the Environment. A higher value gives the Environment more pages, so that there are fewer pages for daemon and user jobs. A lower value gives the Environment fewer pages, so that there are more pages for daemon and user jobs.

The value for Environment_Wsl is an integer from 0 to the number of pages in main memory. However, if the user specifies a value that is too high to leave adequate resources for other jobs, an error message is displayed. The default value for Environment_Wsl is 11,000.

`Daemon_Wsl`

Specifies the working set limit for jobs started by the system daemon (see package Daemon). A higher value gives the daemon more pages, so that there are fewer pages for the Environment and user jobs. A lower value gives the daemon fewer pages, so that there are more pages for the Environment and user jobs.

The value for Daemon_Wsl is an integer from 0 to the number of pages in main memory. However, if the user specifies a value that is too high to leave adequate resources for other jobs, an error message is displayed. The default value for Daemon_Wsl is 200.

`Min_Ce_Wsl, Min_Oe_Wsl, Min_Attached_Wsl, Min_Detached_Wsl, Min_Server_Wsl`

Specifies the minimum working set limit for the following kinds of jobs: core editor (Ce), object editor (Oe), Attached, Detached, and Server, respectively. That is, when a job's working set limit is decreased by Wsl_Decay_Factor, that working set limit cannot fall below the value set by the appropriate parameter. The value for each parameter also determines the initial working set limit that is given to each kind of job upon creation.

The value for each parameter is an integer number of pages from 1 to the value of the corresponding Max_ parameter (see below). For a given job, the Set_Wsl_Limits procedure temporarily overrides the minimum working set limit set by any of these parameters.

`Max_Ce_Wsl, Max_Oe_Wsl, Max_Attached_Wsl, Max_Detached_Wsl, Max_Server_Wsl`

Specifies the maximum working set limit for the following kinds of jobs: core editor (Ce), object editor (Oe), Attached, Detached, and Server, respectively. That is, when a job's working set limit is increased by Wsl_Growth_Factor, that working set limit cannot exceed the value set by the appropriate parameter. These parameters can be used to give preference to some kinds of jobs over others. For example, the default values for Max_Detached_Wsl and Max_Attached_Wsl give background user jobs twice as much memory as foreground user jobs.

The value for Max_Ce_Wsl is an integer number of pages from the value of the corresponding Min_ parameter (see above) to the memory size. For a given job, the Set_Wsl_Limits procedure temporarily overrides the maximum working set limit set by any of these parameters.

`Min_Available_Memory`

Specifies the minimum number of pages that should always be available to the scheduler for dynamic distribution among jobs. If the number of reserve pages falls below the minimum, the scheduler withholds jobs from running.

If the value is large, more pages are reserved and fewer pages are actually being used, possibly resulting in performance decrease. If the value is small, fewer pages are reserved, so that memory is more fully utilized.

The value of Min_Available_Memory is an integer number of pages from 0 through 2,000. The default value for Min_Available_Memory is 1,024.

`Wsl_Decay_Factor`

Specifies the amount by which a user job's working set limit is automatically decreased every 5 seconds.

The value for Wsl_Decay_Factor is an integer number of pages from 0 through 2,000. The default value for Wsl_Decay_Factor is 50.

`Wsl_Growth_Factor`

Specifies the amount by which a user job's working set limit is automatically increased if the job's working set size has exceeded the working set limit during a review interval.

The value for Wsl_Growth_Factor is an integer number of pages from 0 through 2,000. The default value for Wsl_Growth_Factor is 50.

`Page_Withdrawal_Rate`

Specifies the rate at which pages are withdrawn from memory. A higher Page_Withdrawal_Rate means that the scheduler makes more passes through memory during a given period of time, so that pages are withdrawn more frequently. Therefore, when the Page_Withdrawal_Rate is high, the scheduler can charge pages to jobs with more accuracy and can check large jobs for reclaimable pages more frequently. However, the increased scheduler activity may degrade performance.

The value for Page_Withdrawal_Rate is an integer from 0 through 64. This value designates a multiple of the following unit rate: 640 pages per second. When Page_Withdrawal_Rate is 0, no pages are withdrawn. When Page_Withdrawal_Rate is 1 (the default value), then the scheduler passes through memory at a rate of 1 x 640 pages per second. At this rate, the scheduler makes a complete pass every 51.2 seconds.

Parameters for Disk Scheduling

`Disk_Scheduling`

Specifies whether to enable or disable disk scheduling independently of CPU or memory scheduling. The value of Disk_Scheduling is either 0 (disabled) or 1 (enabled). The default value is 1.

`Max_Disk_Load`

Specifies the maximum acceptable disk wait load before the scheduler withholds jobs that are waiting for disk resources. The disk wait load is the number of tasks waiting to read or write a page from disk, averaged over an interval of time and multiplied by 100.

A high Max_Disk_Load value permits increased disk activity; with more jobs waiting on the disk, performance for individual jobs may decrease. A low Max_Disk_Load value restricts disk availability; with fewer jobs waiting on the disk, more jobs are withheld and performance for individual running jobs may improve. Together, the Max_Disk_Load and Min_Disk_Load parameters define a range of acceptable stress on the disks. The wider the range, the less sensitive the scheduler is to changes in the disk wait load.

The value for Max_Disk_Load is a positive integer that must be greater than the value for Min_Disk_Load. The default value for Max_Disk_Load is 250 (that is, an average of 2.5 tasks waiting for disk resources at any given time). See also the Get_Disk_Wait_Load procedure and the Load_Factor subtype.

`Min_Disk_Load`

Specifies the minimum acceptable disk wait load before action is taken. If the disk wait load falls below the limit specified by Min_Disk_Load, the scheduler releases withheld jobs that were waiting for disk resources.

The value for Min_Disk_Load is a positive integer that must be greater than the value for Max_Disk_Load. The default value for Min_Disk_Load is 200 (that is, an average of two tasks waiting for disk resources at any given time). See also the Get_Disk_Wait_Load procedure and the Load_Factor subtype.

Parameters for Background Job Streams

`Foreground_Time_Limit`

Specifies how long, in seconds, an attached job can run before it is allocated background job resources and is subject to queuing on background job streams. To give more preference to foreground jobs, use a higher value. To encourage users to run long jobs in the background, use a lower value.

The default value for Foreground_Time_Limit is 1,800 seconds (30 minutes).

`Background_Streams`

Specifies the number of background job streams. When there are multiple job streams, they are numbered from 1 to the value of Background_Streams.

The default for Background_Streams is 3.

`Stream_Time N`

Specifies the time limit, in minutes, associated with the job stream numbered *N*. Stream_Time limits the time elapsed since a job began running on job stream *N*.

Note that allowing many jobs to remain queued on job streams may cause a disk space shortage. Jobs are more likely to be requeued before they finish if the Stream_Time value for a stream is low, whereas a high Stream_Time value permits jobs to run to completion without being requeued.

The value for each Stream_Time parameter is a number of minutes from 1 through 43,200. The default values for Stream_Times 1 through 3 are 2, 58, and 0 minutes, respectively.

`Stream_Jobs N`

Specifies the number of jobs that can be running on the job stream numbered *N*.

The value for each Stream_Jobs parameter is an integer from 0 through 5. A value of 0 makes sense only if the value of Strict_Stream_Policy is 0 (false). The default values for Stream_Jobs 1 through 3 are 3, 0, and 0, respectively.

`Strict_Stream_Policy`

Specifies whether strict stream policy is in effect (true) or not (false).

The value for Strict_Stream_Policy is either 0 (false) or 1 (true). The default is 0.

References

procedure Display

function Get

procedure Get_Disk_Wait_Load

subtype Load_Factor

---

procedure Set_Job_Attribute

```
procedure Set_Job_Attribute (Job       : Job_Id;
                             Attribute : String  := "Kind";
                             Value     : String  := "Server");
```

Description

Permits the user to change a job attribute.

Parameters

`Job : Job_Id;`

Specifies the number of the job.

`Attribute : String := "Kind";`

Specifies the attribute. Currently the only attribute supported is "Kind", which permits the user to specify which kind of job the current job should be. This is useful in specifying jobs as servers.

`Value : String := "Server";`

Specifies the value for the attribute. The default sets the job kind to a server.

The allowable values are the string representations of the enumerations of the Job_Kind type: Ce, Oe, Attached, Detached, Server, and Terminated.

procedure Set_Wsl_Limits

```
procedure Set_Wsl_Limits (Job       : Job_Id;
                          Min, Max  : Natural);
```

Description

Sets temporary minimum and maximum working set limits that apply only to the specified job.

For the duration of that job, these temporary limits override the limits defined by the relevant scheduler parameters (Min_ and Max_Ce_Wsl, Min_ and Max_Oe_Wsl, and the like), which are described under the Set procedure and in the introduction to this package. Note that, while the job is running, the user can use the Use_Default_Wsl_Limits procedure to revert to the limits that are defined by the scheduler parameters.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Job : Job_Id;`

Specifies the number of the job whose minimum and maximum working set limits the user wants to set.

`Min : Natural;`

Specifies the temporary minimum working set limit for the given job. This value must be greater than 0 and less that the value of Max.

`Max : Natural;`

Specifies the temporary maximum working set limit for the given job. This value must be greater than the value of Min and less than the memory size.

References

procedure Get_Wsl_Limits

procedure Set

procedure Use_Default_Wsl_Limits

procedure State

```
procedure State;
```

Description

Displays the current scheduler state.

Example

The command:

```
   scheduler.state;
```

produces a display such as the following:

```
 JOB    K/S/P  STAT     CPU        CPU    DISK   DISK    WSET  WSET  MAP  RUN
                AGE   SECONDS      MS/S   DW/S   WAITS   SIZE  LIMIT TO   RATIO
------------------------------------------------------------------------------

   4    A/R/O  ++++  2Ø47Ø.825    28.2   13.4   ++++++  11ØØØ  11ØØØ       1.ØØ
   5    A/R/O  ++++  28668.7Ø3   29Ø.8    Ø.Ø     6395    221    2ØØ       1.ØØ
 217    C/I/6  ++++  ØØØ58.912     Ø.Ø    Ø.Ø      299    151    2ØØ       Ø.99
 222   *A/I/Ø  2535  ØØØØ2.872     Ø.Ø    Ø.Ø      1Ø4     15     5Ø 245   Ø.99
 228   *D/I/Ø  398Ø  ØØØ17.132     Ø.Ø    Ø.Ø      178     62     75   Ø   Ø.99
 245    C/I/O  ++++  ØØ354.791     Ø.Ø    Ø.Ø    1Ø1ØØ    121    2ØØ       Ø.91
 253    O/I/6  2829  ØØØ61.517     Ø.Ø    Ø.Ø      773     15     75 245   Ø.91

Run Queue Load      => 2.46, Ø.73, Ø.74, Ø.68
Disk Wait Load      => Ø.26, Ø.74, Ø.56, Ø.55
Withheld Task Load => Ø.ØØ, Ø.Ø1, Ø.Ø1, Ø.Ø1
Available Memory    => 14293
```

The first ten lines of the display list each job by number along with the resources that have currently been allocated to it.

The values that are displayed for each job include:

JOB	Specifies the job number.
K/S/P	Shows the job's Kind, State, and Priority. Values for Kind and State are indicated by their first initial (see Job_Kind type and Job_State type). An asterisk before Kind indicates that the job is being allocated background job resources.
STAT AGE	Shows how long, in tenths of a second, a job has been in its current state. A series of plus signs indicates that the time has exceeded the display's range.

CPU SECONDS	Shows the total amount of CPU time, in seconds, the job has used since it began.
CPU MS/S	Shows the number of milliseconds of CPU time a job has used in the last 5-second evaluation interval.
DISK DW/S	Shows the number of disk waits a job had in the last 5-second evaluation interval.
DISK WAITS	Shows the total number of disk waits a job has had since it began.
WSET SIZE	Shows the job's current working set size, which is the number of pages of memory that the job is using.
WSET LIMIT	Shows the current working set limit on the job's working set size.
MAP TO	Shows what core editor (if any) the job is grouped with for CPU scheduling purposes.
RUN RATIO	Shows the percentage of time the job has run, as compared with the total amount of time the job has either run or been withheld. A value of 1.00 means that the job has never been withheld.

The next three lines of the display (ignoring the blank line) show the various load averages computed over four intervals: the last 100 milliseconds, the last minute, the last 5 minutes, and the last 15 minutes. (These values are also returned by the Get_Run_Queue_Load, Get_Disk_Wait_Load, and Get_Withheld_Task_Load procedures.) The last line displays the number of currently available pages of reserved memory.

References

procedure Get_Disk_Wait_Load

procedure Get_Run_Queue_Load

procedure Get_Withheld_Task_Load

RATIONAL

generic procedure Traverse_Job_Descriptors

This procedure can be used to get a consistent, efficient snapshot of the statistics of one or more jobs.

The formal parameter list of this procedure is:

```
generic
  with procedure Put (Descriptor : Job_Descriptor);
procedure Traverse_Job_Descriptors (First, Last : Job_Id);
```

generic formal procedure Put

```
with procedure Put (Descriptor : Job_Descriptor);
```

Description

Displays the descriptors for the Traverse_Job_Descriptors procedure.

This procedure is called once by the Traverse_Job_Descriptors procedure for each job in the range specified in the call to the Traverse_Job_Descriptors procedure.

Parameters

```
Descriptor : Job_Descriptor;
```

Specifies the information to be put. See the Job_Descriptors type for further information.

procedure Traverse_Job_Descriptors

```
procedure Traverse_Job_Descriptors (First, Last : Job_Id);
```

Description

Calls the Put procedure to display the job descriptors once for each job in the range `First..Last`.

This procedure is used to get a consistent, efficient snapshot of the statistics of one or more jobs. For further information about the information returned, see the Job_Id subtype.

Parameters

```
First : Job_Id;
```

Specifies the number of the first job.

```
Last : Job_Id;
```

Specifies the number of the last job.

procedure Use_Default_Wsl_Limits

```
procedure Use_Default_Wsl_Limits (Job : Job_Id);
```

Description

Cancels the temporary minimum and maximum working set limits that were set by the Set_Wsl_Limits procedure for the specified job.

While the job is running, this procedure allows the user to revert to the limits that are defined by the relevant scheduler parameters (Min_ and Max_Ce_Wsl, Min_ and Max_Oe_Wsl, and the like). These limits are described under the Set procedure and in the introduction to this package.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Job : Job_Id;
```

Specifies the number of the job for which the default minimum and maximum working set limits should be in effect.

References

procedure Get_Wsl_Limits

procedure Set

procedure Set_Wsl_Limits

function Working_Set_Size

```
function Working_Set_Size (Job : Job_Id) return Natural;
```

Description

Returns the number of pages of memory used by the specified job.

Execution of this function requires that the executing job have operator capability.

Parameters

```
Job : Job_Id;
```

Specifies the number of the job.

```
return Natural;
```

Returns the number of pages of memory used by the specified job.

end Scheduler;

RATIONAL

package System_Backup

The procedures in package System_Backup provide a means to save the Environment state on a regular basis. This ensures that the Environment can be restored with minimal loss after a catastrophic system or Environment failure.

The Backup procedure copies the entire Environment onto tape. The three varieties of backup history are ***full, primary,*** and ***secondary***:

- Full backups are complete and self-sufficient. They preserve system information as well as data.
- Primary backups preserve changes made to the Environment since the last full backup.
- Secondary backups preserve changes made to the Environment since the last primary backup.

A backup of any kind produces two kinds of tapes: ***data tapes*** and ***blue tapes***:

- Data tapes contain all data in the Environment.
- Blue tapes contain the system structure.

During startup, the system determines whether recovery is needed and asks whether you want to proceed with recovery. If you proceed with recovery, the system requests the blue (system) tapes so that it can initialize the disks. The system then requests the appropriate data tapes and loads the data.

After a system is recovered from backup tapes, the next backup must be a full backup.

See the ***Rational R1000 Development System: System Manager's Guide*** for more information on backup procedures.

Execution of some of the operations in this package requires that the executing job have operator capability. This is noted in the reference entry if the requirement applies.

procedure Backup

```
procedure Backup (Variety : Kind := System_Backup.Full);
```

Description

Makes a backup of the specified kind (full, primary, or secondary).

The default is full. Note that after a system is recovered from backup tapes, the next backup must be a full backup.

Execution of this procedure requires that the executing job have operator capability.

Parameters

```
Variety :  Kind := System_Backup.Full;
```
Specifies full, primary, or secondary backup. The default is full.

Restrictions

Backups and disk collection cannot be run at the same time.

If you have begun a backup and the disk collection needs to begin, the backup will be terminated by default. This causes the system to wait for the disk collection to begin Waiting_For_Backup_To_Finish phase before initiating the backup. The user can change this default with the procedure !Tools.Disk_Daemon.Set_Backup_Killing (false), which causes the last data tape to be written before disk collection can begin.

If the disk daemon is running and a backup is attempted, disk collection must complete before the backup can begin.

For further information, see the *Rational R1000 Development System: System Manager's Guide.*

Example

The command:

```
   system_backup.backup (full);
```

begins the process for taking a full backup. This command preserves a state of the Environment that can be supplemented with subsequent secondary backups.

---

procedure History

```
procedure History (Entry_Count       : Positive := 10;
                   Full_Backups_Only : Boolean  := False;
                   Tape_Information  : Boolean  := False);
```

Description

Lists the specified number of previous backups.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Entry_Count : Positive := 10;`

Specifies the number of the most recent backups that should be included in the display.

`Full_Backups_Only : Boolean := False;`

Specifies, when true, information on full backups only.

`Tape_Information : Boolean := False;`

Specifies, when true, that the Environment list (in Current_Output) the first data tape involved in each backup.

Example

The command:

```
   system_backup.history;
```

produces a display such as the following for each full, primary, and secondary backup. By default, the command returns information on each of the ten previous backups.

```
   Full Backup 6 Taken At 19-JUN-87 11:19:42
     Blue Tape Vol Name => BACKUP BLUE TAPE, 19-JUN-87 11:19:42 2
     Blue Tape Vol Id   => 002000
     Data Tape Vol Name => BACKUP, 19-JUN-87 11:19:42
     Data Tape Vol Id   => 001900
```

subtype Id

```
subtype Id is Natural;
```

Description

Specifies the identifier assigned to a backup tape during the backup procedures.

Identifiers are listed in the History procedure display.

References

procedure History

type Kind

```
type Kind is (Full, Primary, Secondary);
```

Description

Specifies the kind of backup to be taken with the Backup procedure.

Enumerations

`Full`

Records the complete state of the Environment—a complete backup.

`Primary`

Records changes in the Environment state since the last full backup—an incremental backup.

`Secondary`

Records changes in the Environment since the last primary backup—an incremental backup.

end System_Backup;

package System_Utilities

Package System_Utilities offers a set of capabilities that provide access to various system characteristics. In general, these characteristics cannot be altered with procedures in this package. To change some of the characteristics not alterable by subprograms in this package, use packages Operator, Scheduler, and Terminal (in this book) and package !Commands.Job (documented in SJM). Other characteristics are controlled by the Environment and cannot be explicitly changed.

Package System_Utilities deals with:

- Sessions
- Users
- Terminals
- Jobs
- System hardware

Sessions are created as necessary when a user logs into the system. The Environment creates the session and manages it for the user. One terminal is assigned to each active session. A single user can have more than one active session by logging into the system from more than one terminal and providing a unique session name for each login.

For persons to log in and use the facilities of the Rational Environment, they must have *users* created for them. Users are created by the Environment with the Operator.Create_User procedure. Users have a home library, a username/password pair, and other characteristics that are maintained by the Environment.

Within each session, the user can create multiple jobs to execute programs and commands. Each job is separately scheduled and can have its own scheduler priority and page limits. Page limits are a resource limit that can be established by the system manager to limit the number of virtual memory pages that user jobs can consume. Users can create jobs implicitly when a command window is executed or explicitly with the !Commands.Program.Run_Job and !Commands.Program.Create_Job procedures.

Unless otherwise noted, if illegal values are passed to any of the operations in this package, the Constraint_Error exception is raised.

constant All_Bad_Blocks

```
All_Bad_Blocks : constant Bad_Block_Kinds := 3;
```

Description

Defines a value that indicates all bad disk blocks.

References

function Bad_Block_List

constant Manufacturers_Bad_Blocks

constant Retargeted_Blocks

type Bad_Block_Kinds

```
type Bad_Block_Kinds is new Long_Integer range 0 .. 7;
```

Description

Defines the kinds of bad disk blocks.

References

constant All_Bad_Blocks

function Bad_Block_List

constant Manufacturers_Bad_Blocks

constant Retargeted_Blocks

function Bad_Block_List

```
function Bad_Block_List (For_Volume  : Natural;
                         Kind        : Bad_Block_Kinds  := Retargeted_Blocks)
                                                    return Block_List;
```

Description

Returns a list of bad disk blocks of the specified kind on the specified disk volume.

Parameters

`For_Volume  :  Natural;`

Specifies the volume for which to get the bad block information.

`Kind  :  Bad_Block_Kinds  := Retargeted_Blocks;`

Specifies the kind of bad blocks to get.

`return Block_List;`

Returns the desired list of bad disk blocks. If the volume or kind of bad block is illegal, a null array will be returned.

Errors

If the volume or kind of bad disk block is illegal, a null array will be returned and no exceptions will be raised.

References

constant All_Bad_Blocks

constant Manufacturers_Bad_Blocks

constant Retargeted_Blocks

type Block_List

```
type Block_List is array (Natural range <>) of Integer;
```

Description

Defines the type used to represent a list of bad disk blocks.

References

function Bad_Block_List

subtype Byte_String

```
subtype Byte_String is System.Byte_String;
```

Description

Defines a string of 8-bit bytes.

subtype Character_Bits_Range

```
subtype Character_Bits_Range is Integer range 5 .. 8;
```

Description

Defines the allowed values for the number of bits in a character.

The number of bits in a character depends on the character set that the terminal uses. Normally this is eight bits for the Rational Terminal. Other applications of terminals may require other values.

function Character_Size

```
function Character_Size (Line : Port := System_Utilities.Terminal)
                                          return Character_Bits_Range;
```

Description

Returns the number of bits used for each character on the specified line.

The Character_Bits_Range value can be changed using the Terminal.Set_Character_Size procedure.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line whose number of bits per character is desired. The default returns the number of bits per character for the line attached to the current session.

`return Character_Bits_Range;`

Returns the number of bits per character.

References

procedure Terminal.Set_Character_Size

function Cpu

```
function Cpu (For_Job : Job_Id := System_Utilities.Get_Job)
                                                   return Duration;
```

Description

Returns the CPU time that the specified job has consumed.

This function returns the time that the specified job has used in its execution. By default, the function returns the CPU time of the current job.

Parameters

```
For_Job :  Job_Id := System_Utilities.Get_Job;
```

Specifies the job whose CPU time is desired. The default returns the CPU time of the current job.

```
return Duration;
```

Returns the CPU time.

function Detach_On_Disconnect

```
function Detach_On_Disconnect (Line : Port := System_Utilities.Terminal)
                                                        return Boolean;
```

Description

Determines whether the detach-on-disconnect feature is enabled for the specified line.

See package Terminal for more information on this feature.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

`return Boolean;`

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

Restrictions

This feature is not yet implemented, so the value returned by this function is undefined.

References

procedure Terminal.Set_Detach_On_Disconnect

function Disconnect_On_Disconnect

```
function Disconnect_On_Disconnect
               (Line : Port := System_Utilities.Terminal) return Boolean;
```

Description

Determines whether the disconnect-on-disconnect feature is enabled for the specified line.

See package Terminal for more information on this feature.

Parameters

```
Line :  Port := System_Utilities.Terminal;
```

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

```
return Boolean;
```

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

References

procedure Terminal.Set_Disconnect_On_Disconnect

function Disconnect_On_Failed_Login

```
function Disconnect_On_Failed_Login
                (Line : Port := System_Utilities.Terminal) return Boolean;
```

Description

Determines whether the disconnect-on-failed-login feature is enabled for the specified line.

See package Terminal for more information on this feature.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

`return Boolean;`

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

References

procedure Terminal.Set_Disconnect_On_Failed_Login

function Disconnect_On_Logoff

```
function Disconnect_On_Logoff (Line : Port := System_Utilities.Terminal)
                                                         return Boolean;
```

Description

Determines whether the disconnect-on-logoff feature is enabled for the specified line.

See package Terminal for more information on this feature.

Parameters

```
Line :  Port := System_Utilities.Terminal;
```

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

```
return Boolean;
```

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

References

procedure Terminal.Set_Disconnect_On_Logoff

function Done

```
function Done (Iter : Job_Iterator) return Boolean;
```

Description

Checks whether the iterator has stepped through all of the jobs.

Parameters

```
Iter :  Job_Iterator;
```

Specifies the iterator to be checked.

```
return Boolean;
```

Returns the value true when the iteration is complete; otherwise, the function returns false.

The job iterator filters out inactive jobs when the Value function, the Done function, and the Next procedure are called. Specifically, the Done function yields true only if active jobs remain when it is called, even if jobs active when the Init procedure is called terminate before the Done function is called.

Example

This example demonstrates use of the iteration capability:

```
    Init (Job_Iterator);
    while not Done (Job_Iterator) loop
        ...
        ... Value (Job_Iterator)
        ...
        Next (Job_Iterator);
    end loop;
```

function Done

```
function Done (Iter : Session_Iterator) return Boolean;
```

Description

Checks whether the iterator has stepped through all of the sessions.

Parameters

```
Iter :  Session_Iterator;
```

Specifies the iterator to be checked.

```
return Boolean;
```

Returns the value true when the iteration is complete; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
    Init (Session_Iterator);
    while not Done (Session_Iterator) loop
        ...
        ... Value (Session_Iterator)
        ...
        Next (Session_Iterator);
    end loop;
```

function Done

```
function Done (Iter : Terminal_Iterator) return Boolean;
```

Description

Checks whether the iterator has stepped through all of the terminals.

Parameters

```
Iter : Terminal_Iterator;
```

Specifies the iterator to be checked.

```
return Boolean;
```

Returns the value true when the iteration is complete; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
    Init (Terminal_Iterator);
    while not Done (Terminal_Iterator) loop
        ...
        ... Value (Terminal_Iterator)
        ...
        Next (Terminal_Iterator);
    end loop;
```

function Elapsed

```
function Elapsed (For_Job : Job_Id := System_Utilities.Get_Job)
                                                 return Duration;
```

Description

Returns the elapsed time that the specified job has existed.

This function returns the time that the specified job has been in existence. By default, the function returns the elapsed time of the current job.

Note: The elapsed time for job 4 (the system) is the elapsed time since the machine was booted.

Parameters

`For_Job : Job_Id := System_Utilities.Get_Job;`

Specifies the job whose elapsed time is desired. The default returns the elapsed time of the current job.

`return Duration;`

Returns the elapsed time.

function Enabled

```
function Enabled (Line : Port := System_Utilities.Terminal) return Boolean;
```

Description

Checks whether the specified line is enabled for logging into the Environment.

The Boolean value can be changed by using the Operator.Enable_Terminal procedure.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line to be checked. The default returns the check on the line attached to the current session.

```
return Boolean;
```

Returns the value true when the specified line is enabled for login; otherwise, the function returns false.

References

procedure Operator.Enable_Terminal

function Error_Name

```
function Error_Name
 (For_Session : Session_Id := System_Utilities.Get_Session) return String;
```

Description

Returns the full name of the Standard_Error filename for the indicated session.

If this file is opened using the I/O packages in the Environment, the output goes to the standard error window. This function can be used, for example, in an application that attempts to redirect error output to the standard error window for the job and needs the correct filename to perform the redirection.

Parameters

`For_Session : Session_Id := System_Utilities.Get_Session;`

Specifies the session for which the filename is to be computed. The default is to return the filename for the current session.

`return String;`

Returns the filename.

function Flow_Control

```
function Flow_Control (Line : Port := System_Utilities.Terminal)
                                                           return String;
```

Description

Determines whether software flow control is used for controlling the flow of data transmitted to the device on the specified line.

Flow control is used by some devices to prevent overruns in these devices. Some devices support hardware flow control (CTS receiving, RTS or DTR transmitting) and some devices support software flow control (via XON and XOFF transmissions). Note that hardware flow control via RTS or DTR is not supported for transmission to devices. CTS hardware flow control is available only for transmission.

Legal values returned for methods of transmission flow control are:

```
NONE          XON_XOFF
```

NONE indicates that there is no software flow control for the line and that there may or may not be CTS hardware flow control for the line. Since CTS hardware flow control is enabled or disabled by changing a hardware switch setting in the RS232 distribution panel, the setting of this hardware switch must be used to determine whether hardware flow control is enabled for the line if the value NONE is returned.

XON_XOFF indicates that software flow control is enabled for the line. This value indicates that the Rational system will stop transmitting when it receives an XOFF from the device and may resume transmission when it receives an XON. Note that hardware flow control may also be enabled for the line. Since CTS hardware flow control is enabled or disabled by changing a hardware switch setting in the RS232 distribution panel, the setting of this switch must be used to determine whether hardware flow control is enabled for the line if the value XON_XOFF is returned.

The type of transmission flow control used for the line can be changed by using the Terminal.Set_Flow_Control procedure or by changing a hardware switch setting in the RS232 distribution panel.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies which line is to be checked for flow control. The default returns the flow control method for the line attached to the current session.

```
return String;
```

Returns the method used for controlling the flow of data transmitted to the device. Legal values and their meanings are defined above.

References

procedure Terminal.Set_Flow_Control

---

function Get_Board_Info

```
function Get_Board_Info (Board : Natural) return String;
```

Description

Returns information about the specified hardware circuit board in the system.

Parameters

```
Board : Natural;
```

Specifies the board about which to determine information. The legal values are:

Legal Values

Value	*Board*
0	IOA/IOC
1	SYS
2	SEQ
3	VAL
4	TYP
5	FIU
100	MEM0
101	MEM1
102	MEM2
103	MEM3

```
return String;
```

Returns information about the board.

function Get_Job

```
function Get_Job return Job_Id;
```

Description

Returns the identity of the current job.

Parameters

```
return Job_Id;
```

Returns the identity of the current job—that is, the job that called this function.

procedure Get_Page_Counts

```
procedure Get_Page_Counts (Cache_Pages  : out Natural;
                           Disk_Pages   : out Natural;
                           Max_Pages    : out Natural;
                           For_Job      :     Job_Id    :=
                                  System_Utilities.Get_Job);
```

Description

Returns the virtual memory page counts for the specified job (each page is 1,024 bytes).

The Environment keeps track of the virtual memory pages currently used by a job and compares this count with the Max_Pages resource limit established for the job when new pages are allocated. If the allocation raises the count above the maximum, the Storage_Error exception will be raised. Note that under certain conditions a job may be allowed to allocate more pages than the maximum allowed before the Storage_Error exception is raised.

When a job begins, it is assigned a default page limit. By default, a job is allowed to create 8,000 pages. This default is determined by the value of the Default_Job_Page_Limit session switch. See SJM, Session Switches, for more information on session switches.

When a job elaborates Ada units, this limit may be increased if these units specify larger page limits through the Page_Limit pragma or the Page_Limit library switch. See LM, package Switches, for more information on library switches. The job can change the limit at any time by calling the Set_Page_Limit procedure.

Parameters

`Cache_Pages :  out Natural;`

Returns the number of pages presently in main memory.

`Disk_Pages :  out Natural;`

Returns the number of pages that have disk space allocated for them.

`Max_Pages :  out Natural;`

Returns the limit on the number of pages the job is allowed to create.

```
For_Job : Job_Id := System_Utilities.Get_Job;
```

Specifies the job for which to count the pages. By default, pages will be counted for the job of the caller.

Errors

If the allocation raises the count above the maximum, the Storage_Error exception will be raised. Note that under certain conditions a job may be allowed to allocate more pages than the maximum allowed before the Storage_Error exception is raised.

References

procedure Set_Page_Limit

function Get_Session

```
function Get_Session return Session_Id;

function Get_Session (For_Job : Job_Id) return Session_Id;
```

Description

Returns the session identifier for the job that executed the call to the function or the indicated job.

Parameters

`For_Job : Job_Id;`

Specifies the job for which to determine the session identifier.

`return Session_Id;`

Returns the session identifier for the job that executed the call to the function or the indicated job.

function Home_Library

```
function Home_Library (User : String := User_Name) return String;
```

Description

Returns the full name of the home library for the specified user.

By default, this function returns the home library for the user of the current session.

Parameters

```
User :  String := User_Name;
```

Specifies the simple name of the user for which the home library is to be determined.

```
return String;
```

Returns the full pathname of the home library for the indicated user.

function Image

```
function Image (Version : Directory.Version) return String;
```

Description

Returns the full pathname for the indicated version.

Parameters

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

Specifies the version for which the name is to be computed.

```
return String;
```

Returns the full pathname for the indicated version.

procedure Init

```
procedure Init (Iter : out Session_Iterator);
```

Description

Initializes the session iterator to iterate over all of the sessions that are active—that is, currently logged in.

If one or more sessions are active, the Value function returns the first session using this value of the iterator. If no sessions are active, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter :  out Session_Iterator;
```

Returns the iterator.

Example

This example demonstrates use of the iteration capability:

```
    Init (Session_Iterator);
    while not Done (Session_Iterator) loop
         ...
         ... Value (Session_Iterator)
         ...
         Next (Session_Iterator);
    end loop;
```

procedure Init

```
procedure Init (Iter        : out Job_Iterator;
                For_Session :     Session_Id     := Get_Session);
```

Description

Initializes the job iterator to iterate over all of the jobs that are active for the specified session.

When one or more jobs exist in the session, the Value function returns the first job using this value of the iterator. When no jobs exist in the session, the Done function returns the value true using this value of the iterator.

The job iterator filters out inactive jobs when the Value function, the Done function, and the Next procedure are called. For example, the Value function yields only jobs that are active when it is called, even if jobs active when the Init procedure is called terminate before the Value function is called.

Parameters

```
Iter :  out Job_Iterator;
```

Returns the iterator.

```
For_Session :  Session_Id := Get_Session;
```

Specifies the session for which the iterator is desired. The default returns an iterator for the set of jobs in the current session.

Example

This example demonstrates use of the iteration capability:

```
    Init (Job_Iterator);
    while not Done (Job_Iterator) loop
        ...
        ... Value (Job_Iterator)
        ...
        Next (Job_Iterator);
    end loop;
```

procedure Init

```
procedure Init (Iter : out Terminal_Iterator);
```

Description

Initializes the terminal iterator to iterate over the terminals known to the Environment.

The terminals known to the Environment are those terminal devices that exist in the library !Machine.Devices. When one or more terminals are known, the Value function returns the first terminal using this value of the iterator. When no terminals are known, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter : out Terminal_Iterator;
```

Returns the iterator.

Example

This example demonstrates use of the iteration capability:

```
Init (Terminal_Iterator);
while not Done (Terminal_Iterator) loop
    ...
    ... Value (Terminal_Iterator)
    ...
    Next (Terminal_Iterator);
end loop;
```

function Input_Name

```
function Input_Name
 (For_Session : Session_Id := System_Utilities.Get_Session) return String;
```

Description

Returns the full name of the Standard_Input filename for the indicated session.

If this file is opened using the I/O packages in the Environment, the input comes from the standard input window. This function could be used, for example, in an application that attempts to redirect input to the standard input window for the job and needs the correct filename to perform the redirection.

Parameters

```
For_Session : Session_Id := System_Utilities.Get_Session;
```

Specifies the session for which the filename is to be computed. The default is to return the filename for the current session.

```
return String;
```

Returns the filename.

function Input_Count

```
function Input_Count (Line : Port := System_Utilities.Terminal)
                                                   return Long_Integer;
```

Description

Returns the number of characters input from the specified line since the system was booted.

Input from the line that has not been read by a session or user program is not counted as input.

This function can be used, for example, to create an application that automatically logs out inactive user sessions. Such an application can use the Input_Count and Output_Count functions to determine whether any characters have recently been typed or output on the line for each session.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the input count is desired. The default returns the input count of the line attached to the current session.

```
return Long_Integer;
```

Returns the input count in characters.

References

function Output_Count

function Input_Rate

```
function Input_Rate (Line : Port := System_Utilities.Terminal)
                                                   return String;
```

Description

Returns the input rate of the specified line.

This function returns a string that contains the input rate of the specified line. By default, the function returns the input rate for the line associated with the current session. This value can be changed with the Terminal.Set_Input_Rate procedure.

Legal values for the input rate are:

DISABLE	BAUD_50	BAUD_75	BAUD_110
BAUD_134_5	BAUD_150	BAUD_200	BAUD_300
BAUD_600	BAUD_1200	BAUD_1800	BAUD_2400
BAUD_4800	BAUD_9600	BAUD_19200	EXT_REC_CLK

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the input rate is desired. The default returns the input rate of the line attached to the current session.

```
return String;
```

Returns the input rate. Legal values are defined above.

References

procedure Terminal.Set_Input_Rate

subtype Job_Id

```
subtype Job_Id is Machine.Job_Id range 4 .. 255;
```

Description

Defines a representation for a job.

Objects of the Job_Id subtype are created to represent jobs in the Environment. Jobs are manipulated by procedures in SJM, package Job.

References

SJM, package Job

type Job_Iterator

```
type Job_Iterator is private;
```

Description

Defines a type that allows iterating over all jobs in a specified session.

Objects of the Job_Iterator type contain all of the information necessary to step over all of the jobs in a session. The type is used with the Init and Next procedures and the Value and Done functions.

function Job_Name

```
function Job_Name (For_Job : Job_Id := System_Utilities.Get_Job)
                                                    return String;
```

Description

Returns the symbolic name of the indicated job.

This symbolic name is the name that is put in the job header at the beginning of job output to an output window. Its form is determined by the values of the session switches controlling the format of the job name at the time the owner of the job started it.

If the job is not currently running, the Constraint_Error exception is raised.

Parameters

```
For_Job :  Job_Id := System_Utilities.Get_Job;
```

Specifies the job for which to get the name.

```
return String;
```

Returns the symbolic name of the indicated job.

Errors

The Constraint_Error exception is raised if the job is not currently running.

function Last_Login

```
function Last_Login (User    : String;
                     Session : String := "") return Calendar.Time;
```

Description

Returns the time at which the specified user logged into the specified session.

By default, this function returns the time at which the user logged into any session.

Parameters

```
User : String;
```

Specifies the simple name for the user.

```
Session : String := "";
```

Specifies the simple name for the session. The default value specifies that the last time the user logged into any session is to be computed.

```
return Calendar.Time;
```

Returns the time at which the user logged into the specified session.

function Last_Logout

```
function Last_Logout (User    : String;
                      Session : String := "") return Calendar.Time;
```

Description

Returns the time at which the specified user logged out of the specified session.

By default, this function returns the time at which the user last logged out of any session.

Parameters

```
User : String;
```

Specifies the simple name for the user.

```
Session : String := "";
```

Specifies the simple name for the session. The default value specifies that the last time the user logged out of any session is to be computed.

```
return Calendar.Time;
```

Returns the time at which the user logged out of the specified session.

function Log_Failed_Logins

```
function Log_Failed_Logins (Line : Port := System_Utilities.Terminal)
                                                        return Boolean;
```

Description

Determines whether the log-failed-logins feature is enabled for the specified line.

See package Terminal for more information on this feature.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

`return Boolean;`

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

References

procedure Terminal.Set_Log_Failed_Logins

function Logged_In

```
function Logged_In (User    : String;
                    Session : String := "") return Boolean;
```

Description

Determines whether the specified user is logged into the specified session.

By default, this function determines whether the specified user is logged into any session.

Parameters

`User : String;`

Specifies the simple name for the user.

`Session : String := "";`

Specifies the simple name for the session. The default value specifies that the check should be made to determine whether the user is logged into any session.

`return Boolean;`

Returns true if the user is logged in; otherwise, the function returns false.

function Login_Disabled

```
function Login_Disabled (Line : Port := System_Utilities.Terminal)
                                                        return Boolean;
```

Description

Determines whether login is disabled for the specified line.

When login is disabled—that is, the function returns true—the Operator.Enable_Terminal command fails for the specified line.

See package Terminal for more information on this feature.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

```
return Boolean;
```

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

References

procedure Operator.Enable_Terminal

procedure Terminal.Set_Login_Disabled

function Logoff_On_Disconnect

```
function Logoff_On_Disconnect (Line : Port := System_Utilities.Terminal)
                                                         return Boolean;
```

Description

Determines whether the logoff-on-disconnect feature is enabled for the specified line.

See package Terminal for more information on this feature.

Parameters

`Line :  Port := System_Utilities.Terminal;`

Specifies the line for which the information is desired. The default returns the information for the line attached to the current session.

`return Boolean;`

Returns true if the feature is currently enabled for the line; otherwise, the function returns false.

Restrictions

This feature is not yet implemented.

References

procedure Terminal.Set_Logoff_On_Disconnect

constant Manufacturers_Bad_Blocks

```
Manufacturers_Bad_Blocks  : constant Bad_Block_Kinds  := 1;
```

Description

Defines a value that indicates manufacturer-designated bad disk blocks.

References

function Bad_Block_List

constant Retargeted_Blocks

procedure Next

```
procedure Next (Iter : in out Job_Iterator);
```

Description

Steps the iterator to point to the next job.

When the iterator steps past the last job, the Done function returns the value true.

The job iterator filters out inactive jobs when the Value function, the Done function, and the Next procedure are called. For example, the Value function yields only jobs that are active when it is called, even if jobs active when the Init procedure is called terminate before the Value function is called.

Parameters

```
Iter :  in out Job_Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
    Init (Job_Iterator);
    while not Done (Job_Iterator) loop
        ...
        ... Value (Job_Iterator)
        ...
        Next (Job_Iterator);
    end loop;
```

procedure Next

```
procedure Next (Iter : in out Session_Iterator);
```

Description

Steps the iterator to point to the next session.

When the iterator steps past the last session, the Done function returns the value true.

Parameters

```
Iter :  in out Session_Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
    Init (Session_Iterator);
    while not Done (Session_Iterator) loop
        ...
        ... Value (Session_Iterator)
        ...
        Next (Session_Iterator);
    end loop;
```

procedure Next

```
procedure Next (Iter : in out Terminal_Iterator);
```

Description

Steps the iterator to point to the next terminal.

When the iterator steps past the last terminal, the Done function returns the value true.

Parameters

```
Iter :  in out Terminal_Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
    Init (Terminal_Iterator);
    while not Done (Terminal_Iterator) loop
        ...
        ... Value (Terminal_Iterator)
        ...
        Next (Terminal_Iterator);
    end loop;
```

subtype Object

```
subtype Object is Directory.Object;
```

Description

Defines the representation for an object in the directory system.

function Output_Count

```
function Output_Count (Line : Port := System_Utilities.Terminal)
                                                   return Long_Integer;
```

Description

Returns the number of characters output to the specified line since the system was booted.

Output from the line that has not been read by a session or user program is not counted as output.

This function can be used, for example, to create an application that automatically logs out inactive user sessions. Such an application can use the Output_Count and Input_Count functions to determine whether any characters have recently been typed or output on the line for each session.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the output count is desired. The default returns the output count of the line attached to the current session.

```
return Long_Integer;
```

Returns the output count in characters.

References

function Input_Count

function Output_Name

```
function Output_Name
 (For_Session : Session_Id := System_Utilities.Get_Session) return String;
```

Description

Returns the full name of the Standard_Output filename for the indicated session.

If this file is opened using the I/O packages in the Environment, the output goes to the standard output window. This function can be used, for example, in an application that attempts to redirect output to the standard output window for the job and needs the correct filename to perform the redirection.

Parameters

`For_Session : Session_Id := System_Utilities.Get_Session;`

Specifies the session for which the filename is to be computed. The default is to return the filename for the current session.

`return String;`

Returns the filename.

function Output_Rate

```
function Output_Rate (Line : Port := System_Utilities.Terminal)
                                                     return String;
```

Description

Returns the output rate of the specified line.

This function returns a string that contains the output rate of the specified line. By default, the function returns the output rate for the line associated with the current session. This value can be changed by using the Terminal.Set_Output_Rate procedure.

Legal values for the output rate are:

```
DISABLE       BAUD_50       BAUD_75       BAUD_110
BAUD_134_5    BAUD_150      BAUD_200      BAUD_300
BAUD_600      BAUD_1200     BAUD_1800     BAUD_2400
BAUD_4800     BAUD_9600     BAUD_19200    EXT_REC_CLK
```

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the output rate is desired. The default returns the output rate for the line attached to the current session.

`return String;`

Returns the output rate. Legal values are defined above.

References

procedure Terminal.Set_Output_Rate

function Parity

```
function Parity (Line : Port := System_Utilities.Terminal)
                                                  return Parity_Kind;
```

Description

Returns the kind of parity checking for the specified line.

The Parity_Kind value can be changed by using the Terminal.Set_Parity procedure.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the parity checking is desired. The default returns the parity checking for the line attached to the current session.

`return Parity_Kind;`

Returns the parity kind.

References

procedure Terminal.Set_Parity

type Parity_Kind

```
type Parity_Kind is (None, Even, Odd);
```

Description

Defines the allowed values for the kind of parity checking performed by the system and terminal.

The reliability of communications between the system and the terminal can be determined by parity checking on each character. The parity checking must be performed in the same way by both the terminal and the system. Terminals often perform parity checking in only one way or have a switch to select the kind of parity checking. The Parity_Kind type and the Terminal.Set_Parity procedure are used to make the system perform the same checks as the terminal.

Enumerations

Even

Specifies that even parity checking be performed. This means that all bits transmitted for a single character, including the parity bit, will be an even number of 1s.

None

Specifies that no parity checking be performed.

Odd

Specifies that odd parity checking be performed. This means that all bits transmitted for a single character, including the parity bit, will be an odd number of 1s.

References

procedure Terminal.Set_Parity

subtype Port

```
subtype Port is Natural range 0 .. 4 * 16 * 16;
```

Description

Defines a representation for the terminal ports on the system.

Each terminal is connected to a specific port on the system; the specific port is assigned a value of this type. For each port there is a Terminal object in the !Machine.Devices library with the name Terminal_n, where *n* is the port number of the device.

Ports in the range 16..80 identify physical RS232 ports. Consecutive numbers identify adjacent RS232 connectors on the RS232 distribution panel. Port 16 is the upper-left port. Numbers in the range 240..255 identify Telnet connections that do not correspond to physical ports. Incoming Telnet connections are assigned numbers in a rotary fashion. Note that the existence of these ports is determined by the number of RS232 lines installed on the Rational system and by the presence or absence of the Rational Networking—TCP/IP product.

Port numbers are used in a number of Environment commands—for example, Flow_Control, Operator.Enable_Terminal, Operator.Force_Logoff, and so on. Port numbers are shown for logged in users in the table produced by the What.Users (SJM) command and others. Ports can be opened for I/O by opening objects named !Machine.Devices.Terminal_n, where *n* is a port number.

function Priority

```
function Priority (For_Job : Job_Id := System_Utilities.Get_Job)
                                                   return Natural;
```

Description

Returns the priority of the specified job.

This function returns a priority value for the specified job. The priorities are defined in SJM, package Job, which also has subprograms for changing the priority of a job. By default, the function returns the priority of the current job.

Parameters

```
For_Job :  Job_Id := System_Utilities.Get_Job;
```

Specifies the job for which priority is desired. The default returns the priority of the current job.

```
return Natural;
```

Returns the priority. The allowed range is defined in SJM, package Job.

References

SJM, package Job

function Receive_Flow_Control

```
function Receive_Flow_Control (Line : Port := System_Utilities.Terminal)
                                                          return String;
```

Description

Determines what method, if any, is used for controlling the flow of data received from the device on the specified line.

Flow control is used by some devices to prevent overruns in these devices. Some devices support hardware flow control (CTS receiving, RTS or DTR transmitting) and some devices support software flow control (via XON and XOFF transmissions). Note that hardware flow control via RTS or DTR is not supported for transmission to devices. CTS hardware flow control is available only for transmission.

Legal values for methods of flow control for received data are:

```
NONE        XON_XOFF      RTS         DTR
```

NONE indicates that there is no software or hardware flow control for the line.

XON_XOFF indicates that software flow control is enabled for the line. This value indicates that the device should stop transmitting when it receives an XOFF from the Rational system and can resume transmission when it receives an XON.

RTS indicates that hardware flow control based on the RTS modem control signal is enabled for the line. This value indicates that the device should stop transmitting when the Rational system switches the RTS modem control signal to OFF and can resume transmission when the signal is switched to ON.

DTR indicates that hardware flow control based on the DTR modem control signal is enabled for the line. This value indicates that the device should stop transmitting when the Rational system switches the DTR modem control signal to OFF and can resume transmission when the signal is switched to ON.

The method of flow control for the line can be changed by the Terminal.Set_Receive_Flow_Control procedure.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line to be checked for flow control. The default returns the method of flow control for the line attached to the current session.

```
return String;
```

Returns the method used for controlling the flow of data received from the device. Legal values and their meanings are defined above.

References

procedure Terminal.Set_Receive_Flow_Control

---

function Receive_Xon_Xoff_Bytes

```
function Receive_Xon_Xoff_Bytes (Line : Port := System_Utilities.Terminal)
                                                 return Byte_String;
```

Description

Returns the two-character byte string that contains the Xon and Xoff characters for the receive side of the specified line.

This function returns a byte string with two characters in it. The Xon character, the first byte, enables transmitting. The Xoff character, the second byte, disables transmitting. These characters are used in the absence of hardware flow control in the terminals, and they vary with different terminal types. The characters can be changed by using the Terminal.Set_Receive_Xon_Xoff_Bytes procedure.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the Xon and Xoff characters are desired. The default returns the characters for the line attached to the current session.

```
return Byte_String;
```

Returns the two characters.

References

procedure Terminal.Set_Receive_Xon_Xoff_Bytes

function Receive_Xon_Xoff_Characters

```
function Receive_Xon_Xoff_Characters
                   (Line : Port := System_Utilities.Terminal)  return String;
```

Description

Returns the two-character string that contains the Xon and Xoff characters for the receive side of the specified line.

This function returns a string with two characters in it. Xon, the first character, enables transmitting. Xoff, the second character, disables transmitting. These characters are used in the absence of hardware flow control in the terminals, and they vary with different terminal types. The characters can be changed by using the Terminal.Set_Receive_Xon_Xoff_Characters procedure.

Parameters

```
Line :  Port := System_Utilities.Terminal;
```

Specifies the line for which the Xon and Xoff characters are desired. The default returns the characters for the line attached to the current session.

```
return String;
```

Returns the two characters.

References

procedure Terminal.Set_Receive_Xon_Xoff_Characters

constant Retargeted_Blocks

```
Retargeted_Blocks  : constant Bad_Block_Kinds  := 2;
```

Description

Defines a value that indicates retargeted disk blocks.

References

constant All_Bad_Blocks

function Bad_Block_List

constant Manufacturers_Bad_Blocks

function Session

```
function Session (For_Session : Session_Id := System_Utilities.Get_Session)
                                                          return Version;

function Session (For_Session : Session_Id := System_Utilities.Get_Session)
                                                           return Object;
```

Description

Returns the version or object that is the specified session.

By default, the function returns the value for the current session.

Parameters

`For_Session : Session_Id := System_Utilities.Get_Session;`

Specifies the session for which the version or object is desired. The default returns the value for the current session.

`return Version;`

Returns the version.

`return Object;`

Returns the object.

subtype Session_Id

```
subtype Session_Id is Machine.Session_Id;
```

Description

Defines a representation for a user's session.

Objects of the Session_Id subtype are created to represent sessions in the Environment. Sessions are created when a user logs into the system, and they are permanent objects in the user's home library unless they are explicitly deleted or destroyed.

type Session_Iterator

```
type Session_Iterator is private;
```

Description

Defines a type that allows iterating over all sessions currently active in the Environment.

Objects of the Session_Iterator type contain all of the information necessary to step over all of the sessions. The type is used with the Init and Next procedures and the Value and Done functions.

function Session_Name

```
function Session_Name
 (For_Session : Session_Id := System_Utilities.Get_Session) return String;
```

Description

Returns the name of the specified session.

This function returns a string that contains the simple name of the specified session. By default, the function returns the name of the current session.

Parameters

```
For_Session : Session_Id := System_Utilities.Get_Session;
```

Specifies the session for which the name is desired. The default is the name of the current session.

```
return String;
```

Returns the name of the session.

procedure Set_Page_Limit

```
procedure Set_Page_Limit (Max_Pages : Natural;
                           For_Job   : Job_Id   := System_Utilities.Get_Job);
```

Description

Sets the upper limit for the virtual memory pages created by the specified job (each page is 1,024 bytes).

If a job attempts to create additional pages beyond the maximum page limit, the Storage_Error exception will be raised. This exception may not be raised immediately, but in the worst case the job will not be able to create more than twice the maximum page limit before getting a storage error.

When a job begins, it is assigned a default page limit. By default, a job is allowed to create 8,000 pages. This default is determined by the value of the Default_Job_Page_Limit session switch. See SJM, Session Switches, for more information on session switches.

When a job elaborates Ada units, this limit may be increased if these units specify larger page limits through the Page_Limit pragma or the Page_Limit library switch. See LM, package Switches, for more information on library switches. The job can change the limit at any time by calling procedure Set_Page_Limit.

The current page counts for a job can be determined by using the Get_Page_Counts procedure.

Operator capability is required to set the page limit for jobs of users different from the caller.

Note that the limit does not apply to the pages of files that the job is accessing; it applies only to data and stack space used by a job.

Parameters

```
Max_Pages : Natural;
```

Specifies the maximum number of pages that can be created by the job. In some cases, the job can create up to twice the maximum page limit before getting a storage error.

```
For_Job :  Job_Id := System_Utilities.Get_Job;
```

Specifies the job for which to set the page limit. By default, the limit will be set for the job of the caller.

Errors

If a job attempts to create additional pages beyond the maximum page limit, the Storage_Error exception will be raised. This exception may not be raised immediately, but in the worst case the job will not be able to create more than twice the maximum page limit before getting a storage error.

References

procedure Get_Page_Counts

function Stop_Bits

```
function Stop_Bits (Line : Port := System_Utilities.Terminal)
                                                return Stop_Bits_Range;
```

Description

Returns the number of stop bits being used on the specified line.

The Stop_Bits_Range value can be changed by using the Terminal.Set_Stop_Bits procedure.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the number of stop bits is desired. The default returns the stop bit range for the line attached to the current session.

```
return Stop_Bits_Range;
```

Returns the number of stop bits.

References

procedure Terminal.Set_Stop_Bits

subtype Stop_Bits_Range

```
subtype Stop_Bits_Range is Integer range 1 .. 2;
```

Description

Defines the allowed values for the number of stop bits.

The stop bits are part of the electrical protocol for communicating with a terminal. The required number of stop bits is usually determined by the terminal.

function System_Boot_Configuration

```
function System_Boot_Configuration return String;
```

Description

Returns the name of the Environment software configuration that the system is running.

This is the same name that is printed in the Message window when users log into the Environment.

Parameters

```
return String;
```

Returns the name of the software configuration.

function System_Up_Time

```
function System_Up_Time return Calendar.Time;
```

Description

Returns the time when the system was last booted.

Parameters

```
return Calendar.Time;
```

Returns the time when the system was last booted.

subtype Tape

```
subtype Tape is Natural range 0 .. 4;
```

Description

Defines the possible tape drives for the system.

function Tape_Name

```
function Tape_Name (Drive : Tape := 0) return String;
```

Description

Returns the pathname of the tape object in the library system associated with the specified drive.

Parameters

```
Drive : Tape := 0;
```

Specifies the tape drive for which the pathname is desired.

```
return String;
```

Returns the pathname of the tape object associated with the specified tape drive.

function Terminal

```
function Terminal (For_Session : Session_Id := System_Utilities.Get_Session)
                                                               return Port;

function Terminal (For_Session : Session_Id := System_Utilities.Get_Session)
                                                            return Version;

function Terminal (For_Session : Session_Id := System_Utilities.Get_Session)
                                                             return Object;
```

Description

Returns the port number, the version of the port, or the port object to which the specified session is connected.

By default, the function returns the value for the current session.

Each terminal is connected to a specific port on the system. For each port there is a Terminal object in the !Machine.Devices library with the name Terminal_n, where *n* is the port number of the device.

Ports in the range 16..80 identify physical RS232 ports. Consecutive numbers identify adjacent RS232 connectors on the RS232 distribution panel. Port 16 is the upper-left port. Numbers in the range 240..255 identify Telnet connections that do not correspond to physical ports. Incoming Telnet connections are assigned numbers in a rotary fashion. Note that the existence of these ports is determined by the number of RS232 lines installed on the Rational system and by the presence or absence of the Rational Networking—TCP/IP product.

Port numbers are used in a number of Environment commands—for example, Flow_Control, Operator.Enable_Terminal, Operator.Force_Logoff, and so on. Port numbers are shown for logged in users in the table produced by the What.Users (SJM) command and others. Ports can be opened for I/O by opening objects named !Machine.Devices.Terminal_n, where *n* is a port number.

Parameters

`For_Session : Session_Id := System_Utilities.Get_Session;`

Specifies the session for which the port number, version, or object is desired. The default returns the value for the current session.

`return Port;`

Returns the port number.

```
return Version;
```

Returns the version of the port.

```
return Object;
```

Returns the port object.

---

type Terminal_Iterator

```
type Terminal_Iterator is private;
```

Description

Defines a type that allows iterating over all terminals currently connected to the system.

Objects of the Terminal_Iterator type contain all of the information necessary to step over all of the terminals. The type is used with the Init and Next procedures and the Value and Done functions.

function Terminal_Name

```
function Terminal_Name (Line : Port := System_Utilities.Terminal)
                                                  return String;
```

Description

Returns the full directory name of the terminal object for the specified line.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the terminal name is desired. The default returns the terminal name for the current session's terminal.

```
return String;
```

Returns the full directory name of the terminal object.

function Terminal_Type

```
function Terminal_Type (Line : Port := System_Utilities.Terminal)
                                                   return String;
```

Description

Returns the type of terminal attached to the specified line.

This function returns a string that contains the name of the terminal type that is attached to a specified terminal line. By default, the function returns the type of the terminal attached to the current session. Supported terminal types include Rational, VT100, and Facit. This value can be changed by using the Terminal.Set_Terminal_Type procedure.

Parameters

```
Line : Port := System_Utilities.Terminal;
```

Specifies the line for which the terminal type is desired. The default returns the terminal type for the current session's terminal.

```
return String;
```

Returns the terminal type. Currently allowed values are Rational, VT100, and Facit.

References

procedure Terminal.Set_Terminal_Type

function User

```
function User (For_Session : Session_Id := System_Utilities.Get_Session)
                                                          return Version;

function User (For_Session : Session_Id := System_Utilities.Get_Session)
                                                           return Object;
```

Description

Returns the version or object that represents the user for the specified session.

By default, the function returns the user for the current session.

Parameters

`For_Session : Session_Id := System_Utilities.Get_Session;`

Specifies the session for which the user is desired. The default returns the user for the current session.

`return Version;`

Returns the version.

`return Object;`

Returns the object.

function User_Name

```
function User_Name
 (For_Session : Session_Id := System_Utilities.Get_Session) return String;
```

Description

Returns the name of the user who created the specified session.

This function returns a string that contains the simple name of the user who created the specified session. By default, the function returns the username for the current session.

Parameters

`For_Session : Session_Id := System_Utilities.Get_Session;`

Specifies the session for which the username is desired. The default returns the username for the current session.

`return String;`

Returns the username.

function Value

```
function Value (Iter : Job_Iterator) return Job_Id;
```

Description

Returns the job identifier for the job pointed to by the iterator.

The job iterator filters out inactive jobs when the Value function, the Done function, and the Next procedure are called. Specifically, the Value function yields only jobs that are active when it is called, even if jobs active when the Init procedure is called terminate before the Value function is called.

Parameters

```
Iter : Job_Iterator;
```

Specifies the iteration for which the job identifier is desired.

```
return Job_Id;
```

Returns the job identifier.

Example

This example demonstrates use of the iteration capability:

```
    Init (Job_Iterator);
    while not Done (Job_Iterator) loop
        ...
        ... Value (Job_Iterator)
        ...
        Next (Job_Iterator);
    end loop;
```

function Value

```
function Value (Iter : Session_Iterator) return Session_Id;
```

Description

Returns the session identifier for the session pointed to by the iterator.

Parameters

```
Iter : Session_Iterator;
```

Specifies the iteration for which the session identifier is desired.

```
return Session_Id;
```

Returns the session identifier.

Example

This example demonstrates use of the iteration capability:

```
    Init (Session_Iterator);
    while not Done (Session_Iterator) loop
        ...
        ... Value (Session_Iterator)
        ...
        Next (Session_Iterator);
    end loop;
```

function Value

```
function Value (Iter : Terminal_Iterator) return Natural;
```

Description

Returns the number of the terminal for the terminal pointed to by the iterator.

Parameters

```
Iter : Terminal_Iterator;
```

Specifies the iteration for which the terminal number is desired.

```
return Natural;
```

Returns the terminal number.

Example

This example demonstrates use of the iteration capability:

```
    Init (Terminal_Iterator);
    while not Done (Terminal_Iterator) loop
        ...
        ... Value (Terminal_Iterator)
        ...
        Next (Terminal_Iterator);
    end loop;
```

subtype Version

```
subtype Version is Directory.Version;
```

Description

Defines a representation for a version of an object.

All entities in the Environment are managed by the Environment as a value of this subtype. Each user, job, session, terminal, Ada unit, file, and the like is maintained in the Environment in this way.

function Xon_Xoff_Bytes

```
function Xon_Xoff_Bytes (Line : Port := System_Utilities.Terminal)
                                                    return Byte_String;
```

Description

Returns the two-character byte string that contains the Xon and Xoff characters for the transmit side of the specified line.

This function returns a byte string with two characters in it. The Xon character, the first byte, enables transmitting. The Xoff character, the second byte, disables transmitting. These characters are used in the absence of hardware flow control in the terminals, and they vary with different terminal types. The characters can be changed by using the Terminal.Set_Xon_Xoff_Bytes procedure.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the Xon and Xoff characters are desired. The default returns the characters for the line attached to the current session.

`return Byte_String;`

Returns the two characters.

References

procedure Terminal.Set_Xon_Xoff_Bytes

function Xon_Xoff_Characters

```
function Xon_Xoff_Characters (Line : Port := System_Utilities.Terminal)
                                                          return String;
```

Description

Returns the two-character string that contains the Xon and Xoff characters for the transmit side of the specified line.

This function returns a string with two characters in it. Xon, the first character, enables transmitting. Xoff, the second character, disables transmitting. These characters are used in the absence of hardware flow control in the terminals, and they vary with different terminal types. The characters can be changed by using the Terminal.Set_Xon_Xoff_Characters procedure.

Parameters

`Line : Port := System_Utilities.Terminal;`

Specifies the line for which the Xon and Xoff characters are desired. The default returns the characters for the line attached to the current session.

`return String;`

Returns the two characters.

References

procedure Terminal.Set_Xon_Xoff_Characters

end System_Utilities;

RATIONAL

package Tape

Package Tape defines procedures for tape input and output.

The Read procedure reads tapes written in standard ANSI format. Command parameters allow the option of specifying R1000® and DEC™ VAX™/VMS™ formatted tapes. The Write procedure writes tapes in standard ANSI format. Currently, tapes can be formatted for either the R1000 or VAX/VMS. The Write procedure uses the Format parameter to specify tape format. The Format parameter requires the same syntax as the Options parameter. For further information on specifying options, see the Key Concepts.

Files can be formatted on the R1000 for transfer to DEC VAX/VMS systems. The Environment maps R1000 object names to legal VAX/VMS filenames, and it includes a file called INDEX that lists the object names with their corresponding VAX/VMS names.

Files, including Ada program units, that are read into the R1000 are treated as text files. Ada source files from foreign hosts can be parsed and restored to the Environment as Ada units using the !Commands.Compilation.Parse procedure.

procedure Display_Tape

```
procedure Display_Tape (Drive             : Natural  := 0;
                        Marks_To_Skip     : Integer  := 0;
                        Records_To_Skip   : Integer  := 0;
                        Blocks_To_Display : Natural  := 10);
```

Description

Produces a hexadecimal display of the contents of the specified portion of the tape.

The procedure displays in the current output window the specified portion of the tape. The procedure can specify what area, relative to the tape's current position, to display. Marks and records can be specified either forward from the current position (positive parameter values) or backward from the current position (negative parameter values).

Unless the tape drive is already allocated to your session, the Environment prompts at the operations console for mounting the tape when the command is executed. For further information, see the *Rational R1000 Development System: System Manager's Guide.*

Parameters

`Drive : Natural := 0;`

Specifies the tape drive whose tape is to be displayed. The default is drive 0.

`Marks_To_Skip : Integer := 0;`

Specifies the number of tape marks to skip. Negative values move the tape backward and positive values move the tape forward. The default is to skip no marks.

`Records_To_Skip : Integer := 0;`

Specifies the number of tape records to skip. Negative values move the tape backward and positive values move the tape forward. The default is to skip no records.

`Blocks_To_Display : Natural := 10;`

Specifies the number of tape blocks to display. The default is 10 blocks.

Restrictions

The Environment supports a maximum tape block size of 4 Kb.

exception Error

```
Error : exception;
```

Description

Defines an exception that is raised by several procedures in this package when errors occur.

procedure Examine_Labels

```
procedure Examine_Labels (Vol_Id              : String  := "";
                          Vol_Set_Name        : String  := "";
                          To_Operator         : String  := "Thank You";
                          Volume_Labels_Only  : Boolean := True);
```

Description

Displays the labels on the specified tape volume.

The labels on the tape are displayed in the current output window. The tape must be an ANSI-formatted tape.

Unless the tape drive is already allocated to your session, the Environment prompts at the operations console for mounting the tape when the command is executed. For further information, see the ***Rational R1000 Development System: System Manager's Guide.***

Parameters

`Vol_Id : String := "";`

Specifies the volume identifier of the tape to be examined. The default is no identifier specified.

`Vol_Set_Name : String := "";`

Specifies the name of the volume set to be examined. The default is no name specified.

`To_Operator : String := "Thank You";`

Specifies a message to be displayed at the operations console along with the mount request.

`Volume_Labels_Only : Boolean := True;`

Specifies whether to examine only the label of the volume or all labels on the tape. The default is the volume label only.

procedure Format_Tape

```
procedure Format_Tape (Drive  : Natural;
                       Vol_Id : String   := "");
```

Description

Formats the tape on the specified drive with the specified volume identifier.

The procedure builds an empty ANSI tape. The tape is written with the specified volume identifier.

Parameters

`Drive : Natural;`

Specifies the drive whose tape should be formatted.

`Vol_Id : String := "";`

Specifies the volume identifier to write on the tape. The identifier must be no more than a six-character string. The default is no identifier specified.

procedure Read

```
procedure Read (Volume       : String  := "";
                Directory    : String  := "$";
                Format       : String  := "R1ØØØ";
                To_Operator  : String  := "Thank You";
                Add_New_Line : Boolean := True;
                Response     : String  := "<PROFILE>");
```

Description

Reads the tape and copies its contents into the specified library.

Unless the tape drive is already allocated to your session, the Environment prompts at the operations console for mounting the tape when the command is executed. For further information, see the *Rational R1000 Development System: System Manager's Guide.*

Parameters

```
Volume : String := "";
```

Specifies a standard ANSI volume identification: a six-character string. The default is the null string.

```
Directory : String := "$";
```

Specifies the Environment directory destination for the files being read. The default is the enclosing library.

```
Format : String := "R1ØØØ";
```

Specifies whether the tape was formatted for an R1000 (the default) or DEC VAX/VMS. VAX/VMS filenames are converted by changing the dot preceding the extension to an underscore. To specify files in VAX/VMS format, use the value VAX/VMS for the Format parameter.

```
To_Operator : String := "Thank You";
```

Specifies a message to be displayed to the operator after the tape has been loaded.

```
Add_New_Line : Boolean := True;
```

Specifies whether to add a new line character after each record read from the tape. The default is to add a new line character.

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

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

Restrictions

Currently, this procedure reads only text files.

To move binary files between R1000s using tape, use the !Commands.Archive.Save and Archive.Restore procedures.

Example

The command:

```
   tape.read (Volume => "070501", Directory => "!users.thales",
              Format => "vax/vms");
```

reads the tape with the specified ANSI volume label. The files on the tape are read to the directory !Users.Thales. The VAX/VMS parameter instructs the Environment that the tape was written as VAX/VMS-compatible ANSI tape.

References

LM, package Archive

procedure Read_Mt

```
procedure Read_Mt (Drive : Natural := Ø);
```

Description

This procedure is no longer supported by the Environment.

procedure Rewind

```
procedure Rewind (Drive : Natural := Ø);
```

Description

Rewinds the tape unit and leaves it loaded at the beginning-of-tape marker and on-line.

If the tape drive is not allocated to your session, the system will issue a mount request.

Parameters

```
Drive : Natural := Ø;
```

Specifies the drive number. The default is drive 0.

procedure Unload

```
procedure Unload (Drive : Natural := 0);
```

Description

Rewinds and unloads the tape unit, and then takes it off-line.

If the tape drive is not allocated to your session, the system will issue a mount request. Unloading a tape rewinds it and removes it from the tape drive. You would want to do this when you have aborted a job using a tape to unload the tape so someone else can use the tape drive.

Parameters

```
Drive : Natural := 0;
```

Specifies the drive number. The default is drive 0.

procedure Write

```
procedure Write (Files       : String  := "$@";
                 Volume      : String  := "";
                 Format      : String  := "R1000";
                 To_Operator : String  := "Thank You";
                 Text_Files  : Boolean := True;
                 Response    : String  := "<PROFILE>");
```

Description

Copies the specified objects onto a tape as files.

Unless the tape drive is already allocated to your session, the Environment prompts at the operations console for mounting the tape when the command is executed. For further information, see the *Rational R1000 Development System: System Manager's Guide.*

Parameters

`Files : String := "$@";`

Specifies the objects to write onto tape. The default, $@, is to write all objects in the enclosing library. Note that only text files and Ada units will be written.

`Volume : String := "";`

Specifies a standard ANSI volume identification: a six-character string. If the user does not provide a volume identifier, the Environment generates one.

```
Format : String := "R1000";
```

Allows the user to specify tape format information. The Format parameter requires the same syntax as the Options parameter. See the Key Concepts for further information. The following is a list of Format options:

`Block_Length`

Specifies the block length for the tape. The default is 2,048 bytes. See the Format literal for restrictions.

`Format` literal

Specifies the record format for the tape. The default is variable-length records.

`Fixed_Length`	Use the "Format=Fixed_Length" option to write tapes with fixed-length records. The length of the records is specified in the Record_Length option. The Record_Length value must be less than or equal to the block length.
`Variable_Length`	Use the "Format=Variable_Length" option to write tapes with variable-length records. The maximum record length is specified in the Record_Length option. The Record_Length value must be less than or equal to the block length.
`Spanned`	Use the "Format=Spanned" option to write data with inter-record gaps. This enables the user to write records that are greater than the block length. The maximum record size is specified by the Record_Length option.

`Label=`***string***

Allows the user to label the user's own tapes, where *string* is the tape label. The default is to write no label.

`Record_Length`

Specifies the record length for the tape. The default is 512 bytes. See the Format literal for restrictions.

`Target` literal

Specifies whether to format the tape for an R1000 (the default), VAX/VMS, or MV.

`R1000`	Use the "Target=R1000" option to format the tape for an R1000 system.
`MV`	Use the "Target=MV" option to format tapes for an MV.
`VAX/VMS`	Use the "Target=VAX/VMS" option to format tapes for a VAX/-VMS system. When tapes are formatted for VAX/VMS, R1000 filenames are changed to legal VAX/VMS filenames. The Environment includes a file on the tape called INDEX that maps R1000 filenames to the new, legal VAX/VMS filenames.

```
To_Operator : String := "Thank You";
```

Specifies a message to be displayed at the operations console when the tape mount request is made.

```
Text_Files : Boolean := True;
```

Specifies whether the record length is determined by the line length. If true (the default), the end-of-line mark in the file determines the logical record length.

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

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

Restrictions

This procedure writes only text files and Ada units. Others are skipped, and a warning is generated in the log file.

To move binary files between R1000s using tape, use the !Commands.Archive.Save and Archive.Restore procedures.

Example

Given the following library:

```
!Users.Thales.The_Library
    Works : Ada (Pack_Spec);
    Works : Ada (Pack_Body);
      .Get_String    : (Func_Body);
      .Part_1        : (Proc_Body);
```

the following command:

```
tape.write ("!users.thales.the_library.@");
```

writes onto tape all the objects contained in Thales.The_Library. In addition, because the default true is used for the Recursive parameter, all dependent Ada units (in this example, everything in package Works) are included. Each unit included on the tape is written as a separate file. Because all of the default options for the Format parameter are used, the tape written is an R1000 tape with no label, using variable-length records. The maximum record length is 512 bytes and the maximum block length is 2,048 bytes.

References

LM, package Archive

procedure Write_Mt

```
procedure Write_Mt (File     : String  := "<SELECTION>";
                    Indirect : Boolean := True;
                    Drive    : Natural := 0);
```

Description

This procedure is no longer supported by the Environment.

end Tape;

package Terminal

The procedures in package Terminal are used for configuring an asynchronous port or Telnet port to match the requirements and characteristics of a terminal device (such as a printer or terminal) or a modem.

The terminal types currently supported include Rational, VT100, and Facit.

Many procedures act on the Line parameter, which defaults to the current line—that is, the port to which your terminal is connected. In the following material, the terms *port* and *line* are used interchangeably.

Execution of many of the operations in this package requires operator capability. This is noted in the reference entry if the requirement applies.

VT100 Terminal Support

The Rational Environment supports the use of user-defined terminal names. A terminal name represents four characteristics of the terminal device, as described below:

- Input type: Specifies the name that defines the terminal type—for example, PCAT (that is, IBM PC/AT). The input type can have its own *terminal*_Keys and/or *terminal*_Commands procedures, where *terminal* specifies the terminal name. If the input type does not have these procedures specified for it, the procedures corresponding to the output type described below are used. Note that *terminal*_Macros and *terminal*_Init are not inherited from the output type.
- Output type: Specifies the driver to be used for producing output. Currently, this can be Rational, VT100 or Facit.
- Lines: Specifies the number of lines that the input type supports.
- Columns: Specifies the number of columns that the input type supports.

These characteristics are defined in file !Machine.Editor_Data.Terminal_Types. Each line in the file represents a terminal type, as shown below:

input [output] [lines [columns]]

If *"output"* is omitted when a terminal type is defined, *"input"* must correspond to Rational, VT100, or Facit. This feature can be used to redefine a terminal type for your installation or in the terminal login interaction. If *"lines"* or *"columns"* is omitted, the default value for the input type is used. Input type and output type are limited to 20 characters, and they must be separated by spaces. A sample entry in the file is:

```
PCAT VT100 72 120
```

The !Machine.Editor_Data.Terminal_Types file is read at system boot. Thus, at login, the user can specify a terminal type as shown below:

```
enter terminal type (vt100): pcat 24 80
```

Creating Your Own Terminal Type

If you are creating your own terminal type and your own *terminal*_Keys and *terminal*_Commands procedures, you must use the same names in *terminal*_Keys as those specified in *terminal*_Commands. Normally, this is done using the conventions in !Machine.Editor_Data.Visible_Key_Names. The only requirement in any event is that the *terminal*_Commands procedure have the proper name and be installed. Mapping between actual keystrokes and logical keys is done by string match.

When a terminal line is enabled, the Environment attempts to determine which type of terminal is attached to each port by requesting terminal identification using the ANSI standard sequence. The Environment currently uses this information to distinguish between Rational, VT100, and Facit terminal types. Note that the VT100 and Facit terminal types respond identically, so special code was implemented to distinguish between the two types. For your own new terminal types, it is possible to provide recognition sequences by making entries in the file !Machine.Editor_Data.Terminal_Recognition. An entry in this file consists of:

terminal　　　*recognition*

The terminal name must be a valid simple Ada name.

For example, for the VT100 terminal, the entry might be:

```
Vt100 $[?1; C
```

where $ represents Ascii.Esc.

The space after the semicolon in the entry above will match any character. The maximum length of the recognition sequence is 14 characters. Definitions appearing later in the file override earlier ones, so a user specifying a new terminal with the same sequence as the one above would cause the new one to be recognized. This would also mask automatic recognition of Facit terminals.

When you log in, the Rational Environment will interrogate the terminal and will receive a terminal recognition sequence. It will attempt to match that sequence with those stored in !Machine.Editor_Data.Terminal_Recognition. The last entry matched, if any, determines the ***terminal name,*** which provides the terminal name to *terminal*_Keys and *terminal*_Commands.

subtype Character_Bits_Range

```
subtype Character_Bits_Range is System_Utilities.Character_Bits_Range;
```

Description

Specifies the number of bits per character.

The range is 5 through 8.

renamed function Current

```
function Current (S : Machine.Session_Id := Default.Session) return Port
                                       renames System_Utilities.Terminal;
```

Description

Returns the port number of the specified session.

Parameters

```
S : Machine.Session_Id := Default.Session;
```

Specifies the session in question. The S parameter defaults to the current session (the session in which the function is executed).

subtype Parity_Kind

```
subtype Parity_Kind is System_Utilities.Parity_Kind;
```

Description

Specifies none, even, or odd parity.

subtype Port

```
subtype Port is Natural range Ø .. 4 * 16 * 16;
```

Description

Specifies the range of port numbers.

procedure Set_Character_Size

```
procedure Set_Character_Size
         (Line  : Port                  := Terminal.Current;
          To_Be : Character_Bits_Range  := System_Utilities.Character_Size);
```

Description

Specifies the number of data bits per character.

This setting affects both transmitted and received data.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`To_Be : Character_Bits_Range := System_Utilities.Character_Size;`

Specifies the new character size.

References

procedure Set_Parity

procedure Set_Stop_Bits

function System_Utilities.Character_Size

procedure Set_Detach_On_Disconnect

```
procedure Set_Detach_On_Disconnect
                (Line    : Port    := Terminal.Current;
                 Enabled : Boolean := System_Utilities.Detach_On_Disconnect);
```

Description

This procedure is not currently supported.

procedure Set_Disconnect_On_Disconnect

```
procedure Set_Disconnect_On_Disconnect
          (Line    : Port    := Terminal.Current;
           Enabled : Boolean := System_Utilities.Disconnect_On_Disconnect);
```

Description

Enables or disables the Disconnect_On_Disconnect option for a particular port.

By default, this option is disabled on new systems, when a disk-incompatible release of the Environment is installed, or whenever the Environment state stored on disk is lost.

If this option is enabled for a port, the R1000 responds to an incoming disconnect signal received on the port by initiating an outgoing disconnect signal on that port.

For an asynchronous port, an incoming disconnect signal occurs when the R1000 senses the Data Carrier Detect (DCD) turn from ON to OFF. An outgoing disconnect signal occurs when the R1000 toggles Data Terminal Ready (DTR) from ON to OFF for 3 seconds and then back to ON again.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Enabled : Boolean := System_Utilities.Disconnect_On_Disconnect;`

Specifies the new setting of this option. The default is the current setting.

References

function System_Utilities.Disconnect_On_Disconnect

procedure Set_Disconnect_On_Failed_Login

```
procedure Set_Disconnect_On_Failed_Login
        (Line    : Port    := Terminal.Current;
         Enabled : Boolean := System_Utilities.Disconnect_On_Failed_Login);
```

Description

Enables or disables the Disconnect_On_Failed_Login option for a particular port.

By default, this option is disabled on a new system, when a disk-incompatible release of the Environment is installed, or whenever the Environment state stored on disk is lost.

If this option is enabled for a port, the R1000 initiates an outgoing disconnect signal on the port when a user repeatedly fails to log in on that port—for example, by entering an incorrect password or unrecognized username.

For an asynchronous port,.an outgoing disconnect signal occurs when the R1000 toggles Data Terminal Ready (DTR) from ON to OFF for 3 seconds and then back to ON again.

Execution of this procedure requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Enabled : Boolean := System_Utilities.Disconnect_On_Failed_Login;`

Specifies the new setting of this option. The default is the current setting.

References

function System_Utilities.Disconnect_On_Failed_Login

procedure Set_Disconnect_On_Logoff

```
procedure Set_Disconnect_On_Logoff
                (Line    : Port    := Terminal.Current;
                 Enabled : Boolean := System_Utilities.Disconnect_On_Logoff);
```

Description

Enables or disables the Disconnect_On_Logoff option for a particular port.

By default, this option is disabled on a new system, when a disk-incompatible release of the Environment is installed, or whenever the Environment state stored on disk is lost.

If this option is enabled for a port, the R1000 initiates an outgoing disconnect on the port whenever a user logs off a session running on the port.

For an asynchronous port, an outgoing disconnect signal occurs when the R1000 toggles Data Terminal Ready (DTR) from ON to OFF for 3 seconds and then back to ON again.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Enabled : Boolean := System_Utilities.Disconnect_On_Logoff;`

Specifies the new setting of this option. The default is the current setting.

References

function System_Utilities.Disconnect_On_Logoff

procedure Set_Flow_Control

```
procedure Set_Flow_Control
                          (Line   : Port   := Terminal.Current;
                           To_Be  : String := System_Utilities.Flow_Control);
```

Description

Enables software flow control for data transmitted by the R1000 on the specified line.

A device such as a terminal or a printer attached to an R1000 port can control the flow of data transmitted to it from the R1000 in two ways:

- Hardware flow control: The flow of data is stopped when the device turns off the Clear To Send (CTS) modem control signal. The flow is restarted when the CTS signal is turned on.
- Software flow control: The flow of data is stopped when the device sends the Xoff byte or character. The flow is restarted when the Xon byte or character is sent.

This procedure enables or disables only software flow control. Hardware flow control is enabled or disabled by a hardware configuration switch in the R1000 port controller. There is no software interface to enable or disable hardware flow control.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`To_Be : String := System_Utilities.Flow_Control;`

Specifies whether to set software flow control on or off.

References

procedure Set_Xon_Xoff_Bytes

function System_Utilities.Flow_Control

procedure Set_Input_Rate

```
procedure Set_Input_Rate (Line  : Port   := Terminal.Current;
                          To_Be : String := System_Utilities.Input_Rate);
```

Description

Sets the data rate for data received by the R1000 on the specified line.

Valid incoming data rates include:

```
DISABLE      BAUD_50      BAUD_75      BAUD_110
BAUD_134_5   BAUD_150     BAUD_200     BAUD_300
BAUD_600     BAUD_1200    BAUD_1800    BAUD_2400
BAUD_4800    BAUD_9600    BAUD_19200   EXT_REC_CLK
```

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`To_Be : String := System_Utilities.Input_Rate;`

Specifies the new input rate.

References

procedure Set_Output_Rate

function System_Utilities.Input_Rate

procedure Set_Log_Failed_Logins

```
procedure Set_Log_Failed_Logins
                    (Line    : Port    := Terminal.Current;
                     Enabled : Boolean := System_Utilities.Log_Failed_Logins);
```

Description

Enables or disables the Log_Failed_Logins option for a particular port.

If this option is enabled for a port, the R1000 writes an entry to the system error log when a user repeatedly fails to log in on that port.

By default, this option is disabled.

Execution of this operation requires that the executing job have operator capability.

Parameters

```
Line : Port := Terminal.Current;
```

Specifies the line to be affected. The default is the current line.

```
Enabled : Boolean := System_Utilities.Log_Failed_Logins;
```

Specifies the new setting of this option. The default is the current setting.

References

function System_Utilities.Log_Failed_Logins

procedure Set_Login_Disabled

```
procedure Set_Login_Disabled
               (Line     : Port    := Terminal.Current;
                Disabled : Boolean := System_Utilities.Login_Disabled);
```

Description

Enables or disables the Login_Disabled option for a particular port.

By default, this option is disabled—that is, the port can be enabled for login.

If this option is enabled for a port, the port cannot be enabled for login, even if the Operator.Enable_Terminal procedure is used.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Disabled : Boolean := System_Utilities.Login_Disabled;`

Specifies the new setting of this option. The default is the current setting.

References

function System_Utilities.Login_Disabled

procedure Set_Logoff_On_Disconnect

```
procedure Set_Logoff_On_Disconnect
             (Line    : Port    := Terminal.Current;
              Enabled : Boolean := System_Utilities.Logoff_On_Disconnect);
```

Description

Enables or disables the Logoff_On_Disconnect option for a particular port.

By default, this option is disabled on new systems, when a disk-incompatible release of the Environment is installed, or whenever the Environment state stored on disk is lost.

This option is not currently supported. In the future, if this option is enabled for a port, the R1000 will respond to a disconnect received on that port by logging off the session running on the port.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Enabled : Boolean := System_Utilities.Logoff_On_Disconnect;`

Specifies the new setting of this option. The default is the current setting.

References

function System_Utilities.Logoff_On_Disconnect

---

procedure Set_Output_Rate

```
procedure Set_Output_Rate (Line  : Port   := Terminal.Current;
                           To_Be : String := System_Utilities.Output_Rate);
```

Description

Sets the data rate for data transmitted by the R1000 on the specified line.

Valid incoming data rates include:

```
DISABLE       BAUD_50       BAUD_75       BAUD_110
BAUD_134_5    BAUD_150      BAUD_200      BAUD_300
BAUD_600      BAUD_1200     BAUD_1800     BAUD_2400
BAUD_4800     BAUD_9600     BAUD_19200    EXT_REC_CLK
```

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`To_Be : String := System_Utilities.Output_Rate;`

Specifies the new output rate.

References

procedure Set_Input_Rate

function System_Utilities.Output_Rate

procedure Set_Parity

```
procedure Set_Parity (Line  : Port        := Terminal.Current;
                      To_Be : Parity_Kind := System_Utilities.Parity);
```

Description

Sets the parity to none, even, or odd.

This setting affects both transmitted and received data.

Execution of this operation requires that the executing job have operator capability.

Parameters

```
Line :  Port := Terminal.Current;
```

Specifies the line to be affected. The default is the current line.

```
To_Be :  Parity_Kind := System_Utilities.Parity;
```

Specifies the new parity setting.

References

procedure Set_Character_Size

procedure Set_Stop_Bits

function System_Utilities.Parity

procedure Set_Receive_Flow_Control

```
procedure Set_Receive_Flow_Control
                (Line  : Port   := Terminal.Current;
                 To_Be : String := System_Utilities.Receive_Flow_Control);
```

Description

Enables or disables flow control of data received by the R1000.

Flow control is used by some devices to prevent overruns in these devices. Some devices support hardware flow control (CTS receiving, RTS or DTR transmitting) and some devices support software flow control (via XON and XOFF transmissions). Note that hardware flow control via RTS or DTR is not supported for transmission to devices. CTS hardware flow control is available only for transmission.

Four types of receive flow control are available:

- NONE: Indicates that there is no software or hardware flow control for the line.
- XON_XOFF: Indicates that software flow control is enabled for the line. This value indicates that the device should stop transmitting when it receives an XOFF from the Rational system and can resume transmission when it receives an XON.
- RTS: Indicates that hardware flow control based on the RTS modem control signal is enabled for the line. This value indicates that the device should stop transmitting when the Rational system switches the RTS modem control signal to OFF and can resume transmission when the signal is switched to ON.
- DTR: Indicates that hardware flow control based on the DTR modem control signal is enabled for the line. This value indicates that the device should stop transmitting when the Rational system switches the DTR modem control signal to OFF and can resume transmission when the signal is switched to ON.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`To_Be : String := System_Utilities.Receive_Flow_Control;`

Specifies that flow control is set to none, Xon_Xoff, RTS, or DTR on that line.

References

procedure Set_Flow_Control

procedure Set_Receive_Xon_Xoff_Bytes

procedure Set_Receive_Xon_Xoff_Characters

function System_Utilities.Receive_Flow_Control

---

procedure Set_Receive_Xon_Xoff_Bytes

```
procedure Set_Receive_Xon_Xoff_Bytes
(Line     : Port               := Terminal.Current;
 Xon_Xoff : System.Byte_String := System_Utilities.Receive_Xon_Xoff_Bytes);
```

Description

Specifies flow control bytes so that the R1000 can regulate the data it receives on the specified line.

Using bytes instead of characters for flow control allows the complete character set to be reserved for other uses.

Execution of this operation requires that the executing job have operator capability.

Parameters

```
Line : Port := Terminal.Current;
```

Specifies the line to be affected. The default is the current line.

```
Xon_Xoff : System.Byte_String := System_Utilities.Receive_Xon_Xoff_Bytes;
```

Specifies the new flow control bytes. This parameter takes a string consisting of the Xon byte followed by the Xoff byte.

References

procedure Set_Receive_Flow_Control

function System_Utilities.Receive_Xon_Xoff_Bytes

procedure Set_Receive_Xon_Xoff_Characters

```
procedure Set_Receive_Xon_Xoff_Characters
         (Line     : Port   := Terminal.Current;
          Xon_Xoff : String := System_Utilities.Receive_Xon_Xoff_Characters);
```

Description

Specifies the Xon and Xoff characters used to control the flow of data received by the R1000.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Xon_Xoff : String := System_Utilities.Receive_Xon_Xoff_Characters;`

Specifies the new flow control characters. This parameter takes a string consisting of the Xon character followed by the Xoff character.

References

procedure Set_Receive_Flow_Control

function System_Utilities.Receive_Xon_Xoff_Characters

procedure Set_Stop_Bits

```
procedure Set_Stop_Bits
                   (Line  : Port            := Terminal.Current;
                    To_Be : Stop_Bits_Range := System_Utilities.Stop_Bits);
```

Description

Sets the number of stop bits for the Line parameter.

The Stop_Bits_Range is 1 through 2.

This setting affects transmitted data only. The R1000 can always receive data with any number of stop bits.

Execution of this operation requires that the executing job have operator capability.

Parameters

```
Line :  Port := Terminal.Current;
```

Specifies the line to be affected. The default is the current line.

```
To_Be :  Stop_Bits_Range := System_Utilities.Stop_Bits;
```

Specifies the number of stop bits to be transmitted.

References

procedure Set_Character_Size

procedure Set_Parity

function System_Utilities.Stop_Bits

procedure Set_Terminal_Type

```
procedure Set_Terminal_Type
                    (Line  : Port   := Terminal.Current;
                     To_Be : String := System_Utilities.Terminal_Type);
```

Description

Specifies the terminal type.

Supported terminal types include:

- Rational
- VT100
- Facit Twist Model 4440

This information is used by terminal-handling software (for example, the core editor) in generating terminal output data.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line :  Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`To_Be :  String := System_Utilities.Terminal_Type;`

Specifies the new terminal type. This parameter is case-insensitive. The three supported terminals are Rational, VT100, and Facit.

References

function System_Utilities.Terminal_Type

procedure Set_Xon_Xoff_Bytes

```
procedure Set_Xon_Xoff_Bytes
        (Line     : Port               := Terminal.Current;
         Xon_Xoff : System.Byte_String := System_Utilities.Xon_Xoff_Bytes);
```

Description

Specifies the flow control bytes that the R1000 recognizes on the specified line.

This procedure specifies the Xon and Xoff bytes used to control the flow of data transmitted by the R1000.

Using bytes instead of characters for flow control allows the complete character set to be reserved for other uses.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Xon_Xoff : System.Byte_String := System_Utilities.Xon_Xoff_Bytes;`

Specifies the new flow control bytes. This parameter takes a string consisting of the Xon byte followed by the Xoff byte.

References

function System_Utilities.Xon_Xoff_Bytes

procedure Set_Xon_Xoff_Characters

```
procedure Set_Xon_Xoff_Characters
               (Line     : Port   := Terminal.Current;
                Xon_Xoff : String := System_Utilities.Xon_Xoff_Characters);
```

Description

Specifies the flow control characters that the R1000 recognizes on the specified line.

This procedure specifies the Xon and Xoff characters used to control the flow of data transmitted by the R1000.

Execution of this operation requires that the executing job have operator capability.

Parameters

`Line : Port := Terminal.Current;`

Specifies the line to be affected. The default is the current line.

`Xon_Xoff : String := System.Utilities.Xon_Xoff_Characters;`

Specifies the new flow control characters. This parameter takes a string consisting of the Xon character followed by the Xoff character. The default character for Xon is ASCII DC1; the default character for Xoff is ASCII DC3.

References

function System_Utilities.Xon_Xoff_Characters

procedure Settings

```
procedure Settings (Line : Port := Terminal.Current);
```

Description

Displays a summary of the current settings for the Line parameter.

By default, this procedure returns the settings for the current terminal.

Parameters

```
Line : Port := Terminal.Current;
```

Specifies the terminal number.

Example

The command:

```
    terminal.settings (18);
```

displays information such as the following:

```
    Terminal Settings for Port 18
    -----------------------------
    Terminal Type =                     RATIONAL
    Input Baud Rate =                   EXT_REC_CLK
    Output Baud Rate =                  EXT_REC_CLK
    Parity =                            NONE
    Stop_Bits =                          1
    Char_Size =                         CHAR_8
    Flow Control For Transmit Data =    NONE
    Flow Control For Receive Data =     NONE
    Disconnect_On_Disconnect  =         FALSE
    Disconnect_On_Logoff  =             FALSE
    Disconnect_On_Failed_Login  =       FALSE
    Log_Failed_Logins  =                FALSE
    Login_Disabled  =                   FALSE
```

Note that if the value of Flow Control For Transmit Data were XON_XOFF instead of None, then two more entries would appear, one for XON and one for XOFF.

subtype Stop_Bits_Range

```
subtype Stop_Bits_Range is System_Utilities.Stop_Bits_Range;
```

Description

Specifies the number of stop bits for a terminal line.

The range is 1 through 2.

end Terminal;

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.

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