Rational Environment Reference Manual

Appendix F for the R1000 Target

Document Control Number: 8001A-04 (803-002320)

Rev. 1.3, March 1985
Rev. 2, December 1985
Rev. 2.1, July 1986
Rev. 3.0, July 1987 (Delta)

This document subject to change without notice.

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

THIS PRODUCT CONFORMS TO ANSI/MIL-STD-1815A AS DETERMINED BY THE AJPO UNDER ITS CURRENT TESTING PROCEDURES

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

Preface v

Appendix F for the R1000 Target 1

Compilation 1

Unit States 1

Treatment of Generics 2

Installation 2

Incremental Operations on Installed Units 3

Coding 3

Incremental Operations on Coded Units 4

The Predefined Language Environment 4

Package Standard 5

Package System 6

Unchecked_Deallocation Procedure 7

Unchecked_Conversion Function 7

Package Machine_Code 7

Attributes 8

Pragmas 8

Representation Clauses 10

Representation of Objects 10

Length Clauses 12

Enumeration Representation Clauses 12

Record Representation Clauses 12

Address Clauses 12

Interrupts 12

Chapter 14 I/O 12

Limits 14

Index 15

RATIONAL

Preface

Appendix F describes the implementation-dependent features of the Ada® language implementation provided by the Rational Environment™ for the Rational architecture and the R1000 Development System. Unlike the *Rational Environment Reference Manual* in general, this appendix does not discuss one package or a set of packages of resources provided by the Environment. Instead, it discusses the topics that the *Reference Manual for the Ada Programming Language* specifies must be in Appendix F.

A table of contents appears at the front of this appendix and an index at the back of this appendix.

RATIONAL

Appendix F for the R1000 Target

This section of the *Reference Manual for the Ada Programming Language* is Appendix F for the Rational Environment, the Rational architecture, and the R1000 target. This appendix describes the following implementation-dependent features:

- Compilation
- The predefined language environment
- Attributes
- Pragmas
- Representation clauses
- Chapter 14 I/O
- Limits

Compilation

The following sections introduce some of the concepts that underlie the Rational Environment compilation system and provide a summary of the separate compilation rules for Ada units in the Environment.

Unit States

The Rational Environment provides an integrated representation of programs, independent of their compilation state. In the Environment, no distinction is made between source code, object code, or other implementation-dependent representations.

In the Environment, each Ada unit can be in one of four basic states, ranging from archived, the lowest state, to coded, the highest state. Transforming a program to the state in which it can be executed consists of promoting all of its units from the source state (or from the archived state) to the coded state; finally, promoting a command that references the program will execute it. Each of the states is described in more detail below:

- *Archived*: The image of the unit cannot be edited. Units in this state also do not have the definition capability and structure-oriented highlighting that is available to units in the source, installed, and coded states. Units can be put in the archived state to save space.

- *Source*: The image of the unit can be edited. Other units that reference it (in the Ada sense) cannot be in a state higher than the source state.
- *Installed*: The unit has been syntactically and semantically checked according to the definition of the Ada language. Other units can now reference it (in the Ada sense); that is, they can be promoted from the source state to higher states.
- *Coded*: Code has been generated for the unit, and the unit can be executed from a Command window (if the unit is R1000 code).

Treatment of Generics

Because the Rational Environment and the Rational architecture do not depend on macro expansion approaches to compile generics, the specification and the body of a generic are not required to be compiled at the same time. Bodies of generics can be changed without making the instantiations of these generics obsolete.

If the formal part of a generic contains private (or limited private) types, certain additional implicit dependencies among the specification, body, and instantiations of a generic may be introduced (see Section 13.3.2 of the *Reference Manual for the Ada Programming Language*). The effect of these implicit dependencies is described more fully in "Installation," below, and in the discussion of the Must_Be_Constrained pragma in "Pragmas," later in this section.

Installation

Installation ordering rules follow Ada's separate compilation rules. Specs must be installed before their corresponding bodies are installed. Subunits must be installed after their parents are installed. A unit spec must be installed before another unit that refers to it can be installed. Bodies can be changed without making other units that refer to their specification obsolete.

If the formal part of a generic contains private (or limited private) types, certain additional implicit installation dependencies among the specification, body, and instantiations of a generic may be introduced (see Section 13.3.2 of the *Reference Manual for the Ada Programming Language*).

If the specification and body of such a generic are installed, and if the body contains language constructs that would require constrained actuals for the formal private (or limited private) types, instantiations that do not provide constrained actuals for these formals cannot be installed after this point (semantic errors will be generated). If, on the other hand, the specification for such a generic and at least one instantiation with unconstrained actuals for the formals have been installed, the body for the generic cannot then be installed if it contains language constructs that would require constrained actuals (semantic errors will be generated).

The Environment supports the Must_Be_Constrained pragma, which can be used to provide more explicit control over the treatment of generics with formals that are private (or limited private). More information is available in the description of the Must_Be_Constrained pragma in "Pragmas," later in this section.

It is always legal for a generic actual parameter to be a type with discriminants if the discriminants have default values. In generic unit instantiation, the Rational Environment treats such actual parameters as if they were constrained types. This conforms to the requirements of AI–00037 (a ruling by the Ada board on the interpretation of the LRM).

Literal declarations outside the bounds of the Long_Integer type are rejected at installation time. The bounds of Long_Integer are System.Min_Int .. System.Max_Int.

A parameterless function having the same name and type as an enumeration literal (declared in the same scope) is rejected at installation time. This conforms to AI–00330 (a ruling by the Ada board on the interpretation of the LRM).

Incremental Operations on Installed Units

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

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

Incremental insertion, deletion, and editing of stand-alone comment lines is always allowed.

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

Coding

Code is generated for a unit when the body of the unit is promoted to the coded state. Promoting a specification to coded does not result in the generation of any code. Code is generated to elaborate declarations in a specification when the corresponding body is promoted to coded. Promoting a specification to coded results in information being computed about the specification that allows clients to be coded.

Coding order differs in some respects from installation order. A library unit specification must be coded before its body can be coded. Package, generic package, and task subunits are coded before their parents are coded. Subprogram and generic subprogram subunits are coded after their parents are coded. Library unit specifications must be coded before any clients can be coded. A main program body can be coded only after every specification and body in the closure of the main program has been coded. The system may optimize these strict ordering rules when it can make use of information from previous promotions.

Incremental Operations on Coded Units

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

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

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

The Predefined Language Environment

The following material describes the predefined library units (all in the *Rational Environment Reference Manual*, PT): package Standard, package System, the Unchecked_Deallocation procedure, and the Unchecked_Conversion function.

Package Standard

Package Standard defines all of the predefined identifiers in the language.

```
package Standard is

    type Boolean is (False, True);
    for Boolean'Size use 1;

    type Integer       is range -2**31-1 .. 2**31-1;

    type Long_Integer  is range (-2**62 - 2**62) .. (2**62 - 1 + 2**62);
                            --       -2**63        ..        2**63-1

    type Float is digits 15 range (2.0**1023) - (2.0**97) + (2.0**1023)..
                                - ((2.0**1023) - (2.0**97) + (2.0**1023));
                                 --  -1.7977E308 .. 1.7977E308;

    type Character is (Nul, ..., Del);
    for Character use (0, ..., 127);
    for Character'Size use 8;

    package Ascii is ... end Ascii;

    subtype Natural  is Integer range 0 .. Integer'Last;
    subtype Positive is Integer range 1 .. Integer'Last;

    type String is array (Positive range <>) of Character;

    type Duration is delta 2.0**(-15)
                      -- -3.051757812500E-05
                    range -(2.0**32) .. (2.0**32) - (2.0**(-15));
                      -- -4.294967296000E+09  .. 4.294967296000E+09

    Constraint_Error : exception;
    Numeric_Error    : exception;
    Program_Error    : exception;
    Storage_Error    : exception;
    Tasking_Error    : exception;

end Standard;
```

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, package Standard.

Package System

Package System defines various implementation-dependent types, objects, and subprograms.

Other declarations defined in package System are reserved for internal use and are not documented. These declarations should not be required for users of the Rational Environment.

```
package System is

    type Name     is (R1000);

    System_Name   : constant Name := R1000;

    Bit           : constant :=    1;
    Storage_Unit  : constant :=    1 * Bit;

    Word_Size     : constant :=  128 * Bit;
    Byte_Size     : constant :=    8 * Bit;
    Megabyte      : constant := (2 ** 20) * Byte_Size;
    Memory_Size   : constant := 32 * Megabyte;


    -- System-Dependent Named Numbers

    Min_Int       : constant := Long_Integer'Pos (Long_Integer'First);
    Max_Int       : constant := Long_Integer'Pos (Long_Integer'Last);

    Max_Digits    : constant := 15;
    Max_Mantissa  : constant := 63;
    Fine_Delta    : constant := 1.0 / (2.0 ** 63);
    Tick          : constant := 200.0E-9;

    subtype Priority is Integer range 0 .. 5;


    type Byte is new Natural range 0 .. 255;

    type Byte_String is array (Natural range <>) of Byte;
    -- Basic units of transmission/reception to/from IO devices
    -- The following exceptions are raised by Unchecked_Conversion or
       Unchecked_Conversions

    Type_Error : exception;
    Capability_Error  : exception;
    Assertion_Error: exception;
end System;
```

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, package System.

For additional information on the exceptions, see the reference entries in the *Rational Environment Reference Manual*, PT, Unchecked_Conversion function and package Unchecked_Conversions.

Unchecked_Deallocation Procedure

The Unchecked_Deallocation procedure is used to perform unchecked storage deallocation for values designated by access types that are not tasks and do not contain components that are tasks or pointers to tasks.

Its formal parameter list is:

```
generic
  type Object is limited private;
  type Name   is access  Object;
procedure Unchecked_Deallocation(X  : in out Name);
```

The Unchecked_Deallocation procedure assigns null to X and reclaims storage for the object it designates.

Deallocation is not allowed if the designated type of the access type is a task type or an access to a task type or if it contains such subcomponents. When the designated type or its subcomponents is a generic formal or a private type exported from a subsystem specification having a closed private part, it is not possible to determine at compilation time whether deallocation will be performed. The Allows_Deallocation function in !Tools can be used at run time to determine whether deallocation will be performed.

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, package Unchecked_Deallocation.

Unchecked_Conversion Function

The Unchecked_Conversion generic function converts objects of one type to objects of another type.

Its formal parameter list is:

```
generic
  type Source is limited private;
  type Target is limited private;
function Unchecked_Conversion (S : Source) return Target;
```

The Source type is the type of the source object bit pattern that is to be converted to the Target type.

A faster, package version of the Unchecked_Conversion function can be found in the *Rational Environment Reference Manual*, PT, package Unchecked_Conversions.

For additional information and examples, see the reference entries in the *Rational Environment Reference Manual*, PT, Unchecked_Conversion function and package Unchecked_Conversions.

Package Machine_Code

Package Machine_Code is not currently supported.

Attributes

The Environment supports no implementation-dependent attributes other than those defined in Appendix A of the *Reference Manual for the Ada Programming Language*. The following clarifications and restrictions complement the descriptions provided in Appendix A:

- 'Address: This attribute is not supported; any number returned is meaningless.
- 'First_Bit: This attribute is not supported.
- 'Last_Bit: This attribute is not supported.
- 'Position: This attribute is not supported.
- 'Storage_Size: 'Storage_Size is meaningful only when applied to access types or access subtypes, in which case it returns the number of storage units reserved for the collection associated with the base type for the access type or subtype. The value returned by 'Storage_Size is meaningless for task types or task objects.

Pragmas

The Environment supports pragmas for application software development in addition to those defined in Appendix B of the *Reference Manual for the Ada Programming Language*. They are described below, along with additional clarifications and restrictions for the pragmas defined in Appendix B:

- Controlled: Because the implementation does not support automatic garbage collection, this pragma is always implicitly in effect for the R1000 target.
- Disable_Deallocation (X): This pragma is used to disable deallocation for type X, where X is the name of the type for which you want to disable deallocation.
- Enable_Deallocation (X): This pragma is used with the Unchecked_Deallocation generic to enable deallocation for type X, where X is the name of the access type for which you want to reclaim storage. This pragma can also be used on a generic formal to indicate that it should be deallocatable.
- Inline: This pragma currently has no effect for the R1000 target.
- Interface: The Environment does not currently support the execution of other languages on the Rational architecture. To support development of target-dependent software containing this pragma, however, the Environment recognizes the pragma. The effect of this pragma is that a body is implicitly built that will raise the Program_Error exception if the subprogram is executed when the Ignore_Interface_Pragmas library switch is false.
- List: This pragma currently has no effect.
- Loaded_Main: This pragma is generated by the Environment to specify that a unit is a code-only unit. When package Archive (*Rational Environment Reference Manual*, LM) is used to generate a code-only unit, a Main pragma is converted to a Loaded_Main pragma automatically.

- Main: This pragma is used to cause the Environment to preload the object code for the compilation units referenced by a main program. Normally this loading is done when a Command window referencing these units is promoted.

  The pragma takes no parameters and should be placed immediately after the declaration for the specification or the body of the main subprogram. Note that there is a restriction that the parameters to subprograms containing this pragma must be of types defined in package Standard, package System, or any other predefined package in the Environment directory structure provided by Rational.

  The pragma can be placed only after library units. The loading takes place when the body of the main program is promoted to the coded state. For this to occur, all compilation units referenced by the main program must be in the coded state.

  When subsystems are used, the loading of subprograms containing a Main pragma will use the current activity to determine the actual subsystem implementations that will compose the main program. Once the loading has taken place, the execution of the main program can occur without requiring an activity.

  Executing a main program containing this pragma first causes the closure of the library units referenced by the main program to be elaborated. The program is then executed. If there are references in the Command window to units in the closure of the main program other than within the main program, these references will cause their own copy of these units to be elaborated. These elaborated instances will be separate from those of the main program's elaboration.

- Memory_Size: This pragma has no effect.

- Must_Be_Constrained: This pragma is used in a generic formal part to indicate that formal private (and limited private) types must be constrained or need not be constrained.

  This pragma allows programmers to declare explicitly how they intend to use the formals in the specification for a generic. Then the Environment can check that any instantiations of the generic that are installed before the body of the generic is installed are legal.

  The pragma's syntax is:

  ```
  pragma Must_Be_Constrained ([<cond> =>] <type_id>, ...);
  ```

  The condition can be either yes or no and defaults to the previous value (which is initially yes) if omitted. The type identifier must be a formal private (or limited private) type defined in the same formal part as the pragma.

  If the condition value of no is specified, any use in the body that requires a constrained type will be flagged as a semantic error. If yes is specified, any instantiations that contain actuals that require constrained types will be flagged with semantic errors if the actuals are not constrained.

- Open_Private_Part: This pragma is used in conjunction with subsystems to indicate that a subsystem interface has an open private part.

- Optimize: This pragma currently has no effect.

- Pack: All records and arrays are stored packed in the minimum number of bits that they require, unless explicitly overridden by a length representation clause (see "Representation of Objects," below). Thus, this pragma has no effect.

- Page: This pragma is used by the print spooler to cause a new page. The pragma will be the last line on the page. The next line will be printed on the next page.
- Page_Limit (X): This pragma specifies that the page limit for the current job should be no less than X, where X is a number. This pragma overrides the library switch Page_Limit, which overrides the session switch Default_Job_Page_Limit. For a more detailed description, see the reference entries in the *Rational Environment Reference Manual*, SMU, System_Utilities.Get_Page_Counts and System_Utilities.Set_Page_Limit procedures.
- Priority: Priorities can be specified only inside a task or a library main program. If multiple priorities are specified, only the first priority specified is used. The default priority is 2.
- Private_Eyes_Only: This pragma is used in conjunction with subsystems to indicate that items following the pragma in a context clause are required only in the private part of the subsystem interface.
- Shared: This pragma currently has no effect.
- Storage_Unit: The only legal storage unit value for the Rational architecture is 1.
- Suppress: This pragma currently has no effect.
- System_Name: The only legal system name is R1000.

Representation Clauses

The Rational Environment does not currently provide a complete implementation for representation specifications. To facilitate host/target development of target-dependent code containing representation clauses, however, the Environment will optionally compile unsupported representation clauses when the Ignore_Unsupported_Rep_Specs library switch is set to true.

Representation of Objects

The Environment follows some simple rules for representing objects in virtual memory, and these rules can be used to create objects with arbitrary bit images without using representation clauses.

For discrete types as components of structures (records and arrays), the Rational architecture representation will allocate the minimum amount of space to represent the range imposed by the (possibly dynamic) constraints of the applicable subtype, using a two's complement representation that is zero based.

For example:

```
subtype Binary is Integer range 0 .. 1;    -- uses 1 bit

subtype A is Integer range -3 .. 120;      -- uses 8 bits

type B is new Natural range 0 .. 63;       -- uses 6 bits

type C is new Natural range 1022 .. 1023; -- uses 10 bits

type D is (X, Y, Z);                       -- uses 2 bits
                                           --   X => 0
                                           --   Y => 1
                                           --   Z => 2
type E is (X);                             -- uses 0 bits
```

Size representation clauses are supported for all enumeration types that are not declared with two-part declarations. Thus, the above rules can be overridden. A specific example is the representation for the Character type in package Standard, which takes 8 bits instead of 7 because of a size representation clause.

For records without discriminants, the Rational architecture stores the fields in the order specified in the type declaration, using the minimum space required for each field, with no additional Environment-generated fields.

```
type R1 is              -- uses 8+6+1 = 15 bits
    record
        Field_1 : A;
        Field_2 : B;
        Field_3 : Boolean;
    end record;

type R2 is              -- uses 15+1 = 16 bits
    record
        Field_1 : R1;
        Field_2 : Boolean;
    end record;
```

For constrained array types, the Rational architecture stores the elements packed, using the minimum space for each element, with no additional fields.

```
type A1 is array (1..N) of R1;        -- uses 15*N bits
                                      -- N need not be static

type A2 is array (0..10) of Boolean; -- uses 11 bits

type R3 is                            -- uses 15+11+2 = 28 bits
    record
        Field_1 : R1;
        Field_2 : A2;
        Field_3 : D;
    end record;
```

Length Clauses

- 'Size: The Rational architecture supports the 'Size attribute for discrete types only. These types are further limited in that they can have only a single declaration point (that is, they cannot be incomplete or private types). The size specified must be less than or equal to 64.
- 'Storage_Size for collections: The default collection size is 2**24 bits. The storage size for a collection can range from 2**8 to 2**32 bits. The storage size for a collection determines the number of bits required to represent access types for the collection (for example, for collections of the default 2**24 bit size, the number of bits required to store objects of the access type that is associated with this collection is 24). Only types with single declaration points can have storage size specified (that is, they cannot be incomplete or private types).

  Storage sizes for collections must be specified as static expressions.
- 'Storage_Size for tasks: Because each task in the Rational architecture gets its own virtual address space, storage size specifications for tasks are meaningless and, consequently, are not supported.
- 'Small: This length clause is not currently supported.

Enumeration Representation Clauses

No enumeration representation clauses are currently supported.

Record Representation Clauses

No record representation clauses are currently supported.

Address Clauses

No address clauses are currently supported.

Interrupts

Because interrupts do not exist in the Rational architecture, these representation clauses are not needed and, consequently, are not supported.

Chapter 14 I/O

The Environment supports all of the I/O packages defined in Chapter 14 of the *Reference Manual for the Ada Programming Language*, except for package Low_Level_Io, which is not needed. The Environment also provides a number of other I/O packages. The packages defined in Chapter 14, as well as the other I/O packages supported by the Environment, are more fully documented in the *Rational Environment Reference Manual*, Text Input/Output (TIO) and Data and Device Input/Output (DIO).

The following list summarizes the implementation-dependent features of the Chapter 14 I/O packages:

- Filenames: Filenames must conform to the syntax of Ada identifiers. They can, however, be keywords of the Ada language.
- Form parameter: Depending on the external file being written to, this parameter affects the way terminals and Ada units are read. For example, it can specify whether to have the Page pragma read with the Page_Pragma_Mapping option.
- Instantiations of package Direct_Io and package Sequential_Io with access types: Such instantiations are allowed. If files are created or opened using such instantiations, the Use_Error exception is raised.
- Count type: The Count type for package Text_Io and package Direct_Io is defined as:

```
package Text_Io is
  ...
  type Count is range 0 .. 1_000_000_000;
  ...
end Text_Io;

package Direct_Io is
  ...
  type Count is new Integer
              range 0 .. Integer'Last/Element_Type'Size;
  ...
end Direct_Io;
```

- Field subtype: The Field subtype for package Text_Io is defined as:

```
subtype Field is Integer range 0 .. Integer'Last;
```

- Standard_Input and Standard_Output files: When a job is run from a Command window, these files are the interactive input/output windows provided by the Rational Editor. When a job is run from package Program, options allow the user to specify what Standard_Input and Standard_Output will be.
- Internal and external files: More than one internal file can be associated with a single external file for input only. Only one internal file can be associated with a single external file for output or inout.
- Sequential_Io and Direct_Io packages: Package Sequential_Io can be instantiated for unconstrained array types or for types with discriminants without default discriminant values. Package Direct_Io cannot be instantiated for unconstrained array types or for types with discriminants without default discriminant values.
- Terminators: The line terminator is denoted by the character Ascii.Lf, the page terminator is denoted by the character Ascii.Ff, and the end-of-file terminator is implicit at the end of the file. A line terminator directly followed by a page terminator is compressed to the single character Ascii.Ff. The line and page terminators preceding the file terminator are implicit and do not appear as characters in the file. For the sake of portability, programs should not depend on this representation, although it can be necessary to use this representation when importing source from another environment or exporting source from the Rational Environment.

- Treatment of control characters: Control characters, other than the terminators described above, are passed directly to and from files to application programs.
- Concurrent properties: The Chapter 14 I/O packages assume that concurrent requests for I/O resources will be synchronized by the application program making the requests, except for package Text_Io, which will synchronize requests for output.

Limits

The following package specifies the absolute limits on the use of certain language features:

```
with system;
package Limits is

   Large : constant := <some very large number>;

    -- Scanner
   Max_Line_Length                          : constant := 254;

   -- Semantics
   Max_Discriminants_In_Constraint          : constant := 256;
   Max_Associations_In_Record_Aggregate     : constant := 256;
   Max_Fields_In_Record_Aggregate           : constant := 256;
   Max_Formals_In_Generic                   : constant := 256;
   Max_Nested_Contexts                      : constant := 250;
   Max_Nested_Packages                      : constant := Large;
   Max_Units_In_Transitive_Closure_Of_With_Lists
                                            : constant := Large;
                    -- (limited by virtual memory stack size)
   Max_Number_Of_Libraries                  : constant := Large;

   -- Code Generator
   Max_Indices_In_Array_Aggregate           : constant := 64;
   Max_Parameters_In_Call                   : constant := 255;
   Max_Expression_Nesting_Depth             : constant := Large;
                    -- (limited by virtual memory stack size)
   Max_Number_Of_Fields_In_Records          : constant := 255;
   Max_Number_Of_Entries_In_A_Task          : constant := 255;
   Max_Number_Of_Dimensions_In_An_Array     : constant := 63;
   Max_Nesting_Of_Subprograms_Or_Blocks_In_A_Package
                                            : constant := 14;

   -- Execution
   Max_Number_Of_Tasks                      : constant := Large;
                    -- (limited by available disk space)
   Max_Object_Size                          : constant := (2**32)*System.Bit;

end Limits;
```

Index

Italicized page numbers in this index denote the main entries for command descriptions.

A

'Address attribute 8
archived state 1
attributes 8

C

Character type 11
coded state 2
concurrent properties
  I/O packages 14
control characters 14
Controlled pragma 8
Count type 13

D

Direct_Io package 13
Disable_Deallocation pragma 8

E

Enable_Deallocation pragma 8
external files 13

F

Field subtype 13
filenames 13
'First_Bit attribute 8
Form parameter 13

I

Inline pragma 8
installed state 2
Interface pragma 8
internal files 13

L

'Last_Bit attribute 8
length clauses 12
Limits package 14
List pragma 8
Loaded_Main pragma 8
Low_Level_Io package 12

M

Machine_Code package 7
Main pragma 9
Memory_Size pragma 9
Must_Be_Constrained pragma 2, 9

O

Open_Private_Part pragma 9
Optimize pragma 9

P

Pack pragma 9
Page pragma 10
Page_Limit pragma 10
'Position attribute 8
pragmas 8
Priority pragma 10
Private_Eyes_Only pragma 10

R

representation clauses 10

S

Sequential_Io package 13
Shared pragma 10
'Size attribute 12
'Small attribute 12
source state 2
Source type 7
Standard package 4
Standard_Input function 13
Standard_Output function 13
'Storage_Size attribute 8, 12
Storage_Unit pragma 10
Suppress pragma 10
System package 4, 6

System_Name pragma 10

T

terminators 13
Text_Io package 13

U

Unchecked_Conversion function 4
Unchecked_Conversion generic function . . 7
Unchecked_Deallocation procedure . . 4, 7
unit state 1

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