Rational Environment Reference Manual

Programming Tools (PT)

Copyright © 1985, 1986, 1987 by Rational

Document Control Number: 8001A-29

Rev. 1.0, November 1985
Rev. 2.0, April 1986
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.

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

Rational and R1000 are registered trademarks and Rational Environment and Rational Subsystems are trademarks of Rational.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

How to Use This Book xi

generic function Allows_Deallocation 1

function Allows_Deallocation 2
generic formal type Name 3
generic formal type Object 4

end Allows_Deallocation

package Calendar 5

function Clock 6
function Day 6
subtype Day_Duration 6
subtype Day_Number 6
function Month 6
subtype Month_Number 6
function Seconds 6
procedure Split 7
type Time 7
exception Time_Error 7
function Time_Of 7
function Year 7
subtype Year_Number 7
function “+” 8
function “–” 8
function “<” 8
function “<=” 8
function “>” 8
function “>=” 8

end Calendar

generic package Concurrent_Map_Generic 9
function Cardinality 10
procedure Copy 11
procedure Define 12
generic formal type Domain_Type 13
function Done 14
function Eval 15
procedure Find 16
generic formal function Hash 18
procedure Init 19
procedure Initialize 21
function Is_Empty 22
function Is_Nil 23
type Iterator 24
procedure Make_Empty 25
type Map 26
exception Multiply_Defined 27
procedure Next 28
function Nil 29
type Pair 30
generic formal type Range_Type 31
generic formal object Size 32
procedure Undefine 33
exception Undefined 34
function Value 35

end Concurrent_Map_Generic

package Hash 37
function Long_Integer_To_Integer 38
generic function Pointer_To_Integer 39
function Pointer_To_Integer 40
generic formal type Ptr 41
generic formal type T 42
end Pointer_To_Integer
generic function Pointer_To_Long_Integer 43
function Pointer_To_Long_Integer 44

generic formal type Ptr . . . 45
generic formal type T . . . 46
end Pointer_To_Long_Integer

end Hash

generic package List_Generic . . . 49
function Done . . . 50
generic formal type Element . . . 51
function First . . . 52
procedure Free . . . 53
procedure Init . . . 54
function Is_Empty . . . 56
type Iterator . . . 57
function Length . . . 58
type List . . . 59
function Make . . . 60
procedure Next . . . 61
function Nil . . . 62
function Rest . . . 63
procedure Set_First . . . 64
procedure Set_Rest . . . 65
function Value . . . 66

end List_Generic

generic package Map_Generic . . . 67
function Cardinality . . . 68
procedure Copy . . . 69
procedure Define . . . 70
generic formal type Domain_Type . . . 71
function Done . . . 72
function Eval . . . 73
procedure Find . . . 74
generic formal function Hash . . . 76
procedure Init . . . 77
procedure Initialize . . . 79
function Is_Empty . . . 80
function Is_Nil . . . 81
type Iterator . . . 82

procedure Make_Empty 83
type Map 84
exception Multiply_Defined 85
procedure Next 86
function Nil 87
type Pair 88
generic formal type Range_Type 89
generic formal object Size 90
procedure Undefine 91
exception Undefined 92
function Value 93

end Map_Generic

generic package Queue_Generic 95
procedure Add 96
procedure Copy 97
procedure Delete 98
function Done 99
generic formal type Element 100
function First 101
procedure Init 102
procedure Initialize 104
function Is_Empty 105
type Iterator 106
procedure Make_Empty 107
procedure Next 108
type Queue 109
function Value 110

end Queue_Generic

generic package Set_Generic 111
procedure Add 112
procedure Copy 113
procedure Delete 114
function Done 115
generic formal type Element 116
procedure Init 117
procedure Initialize 118

function Is_Empty 119
function Is_Member 120
type Iterator 121
procedure Make_Empty 122
procedure Next 123
type Set 124
function Value 125

end Set_Generic

package Simple_Status 127
type Condition 129
type Condition_Class 130
type Condition_Name 131
procedure Create_Condition 132
function Create_Condition_Name 133
function Display_Message 134
function Equal 135
function Error 136
function Error_Type 138
procedure Initialize 139
function Message 140
function Name 141
function Severity 142

end Simple_Status

generic package Stack_Generic 143
procedure Copy 144
function Done 145
generic formal type Element 146
function Empty 147
constant Empty_Stack 148
procedure Init 149
type Iterator 151
procedure Make_Empty 152
procedure Next 153
procedure Pop 154
procedure Push 155
type Stack 156

function Top 157
exception Underflow 158
function Value 159

end Stack_Generic

package Standard 161

end Standard

package System 163
type Address 164
exception Assertion_Error 164
constant Bit 164
type Byte 164
constant Byte_Size 164
type Byte_String 164
exception Capability_Error 164
constant Fine_Delta 165
constant Max_Digits 165
constant Max_Int 165
constant Max_Mantissa 165
constant Megabyte 165
constant Memory_Size 165
constant Min_Int 166
type Name 166
constant Null_Address 166
subtype Priority 166
constant Storage_Unit 166
constant System_Name 166
constant Tick 167
exception Type_Error 167
constant Word_Size 167

end System

generic procedure Table_Sort_Generic 169
generic formal function "<" 170
generic formal type Element 171
generic formal type Element_Array 172
generic formal type Index 173
procedure Table_Sort_Generic 174

end Table_Sort_Generic

package Time_Utilities 175
function "+" 176
function "-" 177
function Convert 178
function Convert_Time 179
type Date_Format 180
constant Day 181
type Day_Count 182
function Day_Of_Week 183
type Days 184
function Duration_Until 185
function Duration_Until_Next 186
function Get_Time 187
constant Hour 188
type Hours 189
function Image 190
type Image_Contents 192
type Interval 193
function Is_Nil 194
type Military_Hours 195
type Milliseconds 196
constant Minute 197
type Minutes 198
type Months 199
function Nil 200
type Seconds 201
type Sun_Positions 202
type Time 203
type Time_Format 204
function Value 205

type Weekday 206
type Years 207

end Time_Utilities

generic function Unchecked_Conversion 209
generic formal type Source 210
generic formal type Target 211
function Unchecked_Conversion 212

end Unchecked_Conversion

package Unchecked_Conversions 219
generic package Unchecked_Conversion_Package 225
function Convert 226
generic formal type Source 229
generic formal type Target 230
end Unchecked_Conversion_Package
generic function Convert_From_Byte_String 231
function Convert_From_Byte_String 232
generic formal type Target 235
end Convert_From_Byte_String
generic function Convert_To_Byte_String 237
function Convert_To_Byte_String 238
generic formal type Source 240
end Convert_To_Byte_String

end Unchecked_Conversions

generic procedure Unchecked_Deallocation 241
Unchecked Deallocation: R1000 242
generic formal type Name 244
generic formal type Object 245
procedure Unchecked_Deallocation 246

end Unchecked_Deallocation

Index 249

How to Use This Book

The Programming Tools (PT) book of the *Rational Environment Reference Manual* contains reference information describing some of the programming tools provided by the Rational Environment™. These tools include a set of reusable software components and various other useful programming tools, along with reference information on the Ada®-predefined packages Calendar, Standard, and System. Note that tools for manipulating strings are documented in String Tools (ST) 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):

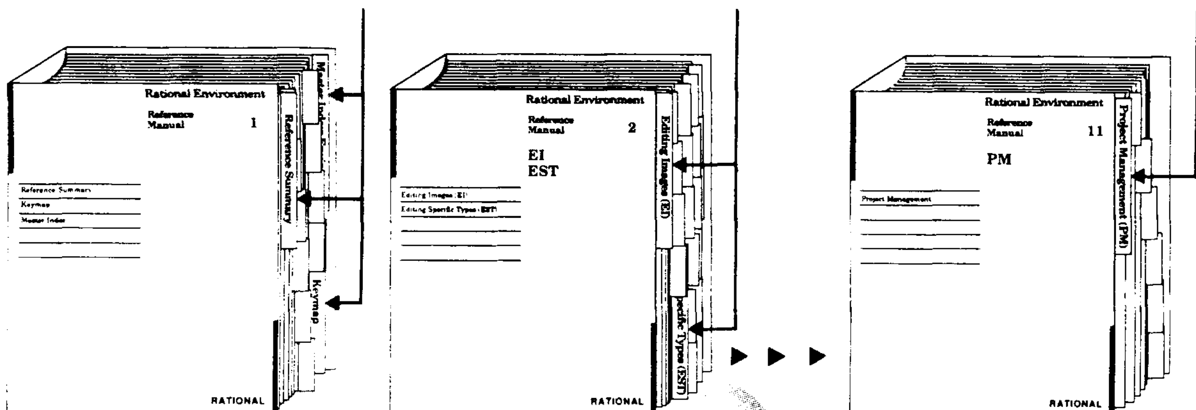
1	Reference Summary Keymap Master Index
2	Editing Images (EI) Editing Specific Types (EST)
3	Debugging (DEB)
4	Session and Job Management (SJM)
5	Library Management (LM)
6	Text Input/Output (TIO)
7	Data and Device Input/Output (DIO)
8	String Tools (ST)
9	Programming Tools (PT)
10	System Management Utilities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

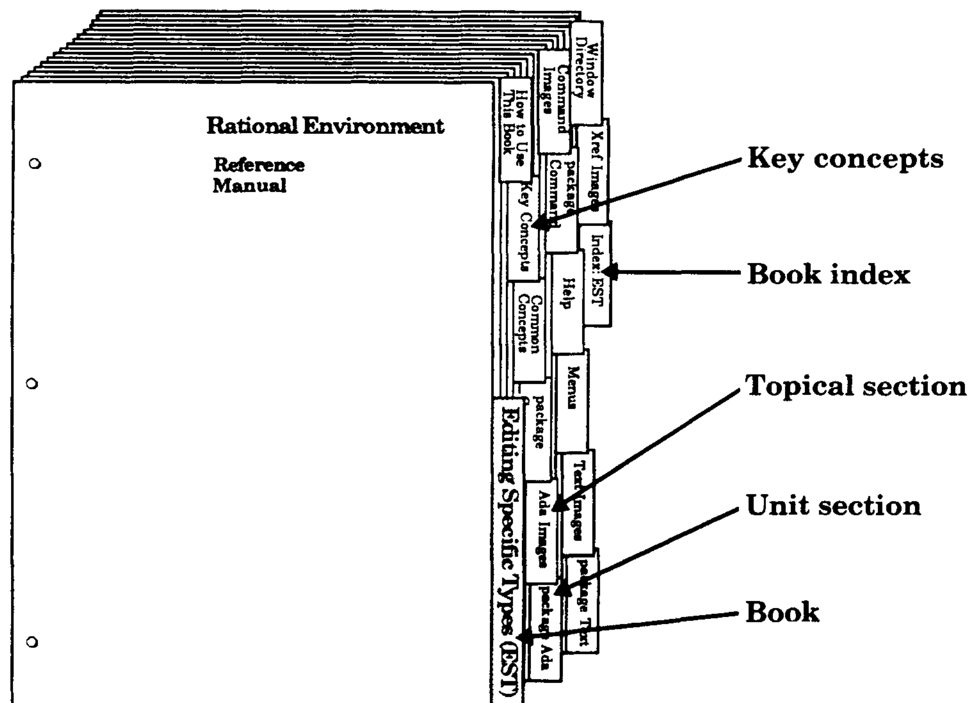
Each *volume* of the Reference Manual contains one or more *books* separated by large colored tabs. Each book contains information on particular features or areas of application in the Environment. The abbreviation for the name of each book (for example, EI for Editing Images) appears on the binder cover 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*

11 volumes containing 14 books

Volume 1: 3 books | Volume 2: 2 books | Volume 11: 1 book





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.

generic function Allows_Deallocation

The Allows_Deallocation generic function determines whether unchecked storage deallocation can be performed on a designated object using the Unchecked_Deallocation generic procedure. Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components.

The formal parameter list of this procedure is:

```
generic
    type Object is limited private;
    type Name is access Object;
function Allows_Deallocation return Boolean;
```

The Object type is the type of the object for which storage is to be reclaimed. The Name type is an access type that points to objects of the Object type.

function Allows_Deallocation

```
function Allows_Deallocation return Boolean;
```

Description

Determines whether unchecked storage deallocation can be performed on objects of Object type using the Unchecked_Deallocation generic procedure.

Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components.

Parameters

```
return Boolean;
```

Returns the value true if storage can be reclaimed for the object; otherwise, the function returns false.

Restrictions

Deallocation is not currently supported for types that contain segmented heap pointers. Many of the types defined by the Environment contain segmented heap pointers and thus cannot be deallocated. Such pointers are not intended for use in application programs and should be used only by Rational technical representatives.

References

generic procedure Unchecked_Deallocation

generic formal type Name

```
type Name is access Object;
```

Description

Defines the pointer to the object on which deallocation will be performed.

Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components.

Restrictions

Deallocation is not currently supported for types that contain segmented heap pointers. Many of the types defined by the Environment contain segmented heap pointeers and thus cannot be deallocated. Such pointers are not intended for use in application programs and should be used only by Rational technical representatives.

generic formal type Object

```
type Object is limited private;
```

Description

Defines the type of object on which deallocation will be performed.

Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components.

Restrictions

Deallocation is not currently supported for types that contain segmented heap pointers. Many of the types defined by the Environment contain segmented heap pointers and thus cannot be deallocated. Such pointers are not intended for use in application programs and should be used only by Rational technical representatives.

References

function Allows_Deallocation

end Allows_Deallocation;

package Calendar

```
package Calendar is
 type Time is private;
 subtype Year_Number  is Integer  range 1901 .. 2099;
 subtype Month_Number is Integer  range 1    .. 12;
 subtype Day_Number   is Integer  range 1    .. 31;
 subtype Day_Duration is Duration range 0.0  .. 86_400.0;

 function Clock return Time;

 function Year    (Date : Time) return Year_Number;
 function Month   (Date : Time) return Month_Number;
 function Day     (Date : Time) return Day_Number;
 function Seconds (Date : Time) return Day_Duration;

 procedure Split (Date    :     Time;
                  Year    : out Year_Number;
                  Month   : out Month_Number;
                  Day     : out Day_Number;
                  Seconds : out Day_Duration);

 function Time_Of (Year    : Year_Number;
                   Month   : Month_Number;
                   Day     : Day_Number;
                   Seconds : Day_Duration := 0.0) return Time;

 function "+" (Left : Time; Right : Duration) return Time;
 function "+" (Left : Duration; Right : Time) return Time;
 function "-" (Left : Time; Right : Duration) return Time;
 function "-" (Left : Time; Right : Time)     return Duration;

 function "<"  (Left, Right : Time) return Boolean;
 function "<=" (Left, Right : Time) return Boolean;
 function ">"  (Left, Right : Time) return Boolean;
 function ">=" (Left, Right : Time) return Boolean;

 Time_Error : Exception;
end Calendar;
```

Description

Ada requires a predefined library package called Calendar. The *Reference Manual for the Ada Programming Language* contains the general specification in Section 9.6, "Delay Statements, Duration, and Time." The Rational implementation specification of package Calendar is given in this section.

```
function Clock return Time;
```

Returns the current clock time.

```
function Day (Date : Time) return Day_Number;
```

Returns the current day of the month from the specified time.

```
subtype Day_Duration is Duration range 0.0 .. 86_400.0;
```

Defines the range of seconds in a day.

```
subtype Day_Number is Integer range 1 .. 31;
```

Defines the range of days in a month.

```
function Month (Date : Time) return Month_Number;
```

Returns the current month from the specified time.

```
subtype Month_Number is Integer range 1 .. 12;
```

Defines the range of months in a year.

```
function Seconds (Date : Time) return Day_Duration;
```

Returns the current second of the day from the specified time.

```
procedure Split (Date    :     Time;
                 Year    : out Year_Number;
                 Month   : out Month_Number;
                 Day     : out Day_Number;
                 Seconds : out Day_Duration);
```

Splits the specified time into year, month, day, and seconds.

```
type Time is private;
```

Defines the basic type for handling time.

```
Time_Error : exception;
```

Defines an exception that can be raised by several functions in this package.

This exception is raised by the Time_Of function if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "−" if, for the given operands, these operators cannot return a date whose year number is in the range of the corresponding subtype, or if the operator "−" cannot return a result that is in the range of the Duration type.

```
function Time_Of (Year    : Year_Number;
                  Month   : Month_Number;
                  Day     : Day_Number;
                  Seconds : Day_Duration := 0.0) return Time;
```

Converts a year, month, day, and seconds into a time.

```
function Year (Date : Time) return Year_Number;
```

Returns the current year from the specified time.

```
subtype Year_Number is Integer range 1901 .. 2099;
```

Defines the range of allowable years.

```
function "+" (Left : Time; Right : Duration) return Time;
function "+" (Left : Duration; Right : Time) return Time;
```

Returns the sum of the specified time and duration.

```
function "-" (Left : Time; Right : Duration) return Time;
function "-" (Left : Time; Right : Time) return Duration;
```

Returns the difference of the specified time and duration or specified times.

```
function "<" (Left, Right : Time) return Boolean;
```

Determines whether one specified time is less than another specified time.

```
function "<=" (Left, Right : Time) return Boolean;
```

Determines whether one specified time is less than or equal to another specified time.

```
function ">" (Left, Right : Time) return Boolean;
```

Determines whether one specified time is greater than another specified time.

```
function ">=" (Left, Right : Time) return Boolean;
```

Determines whether one specified time is greater than or equal to another specified time.

---

generic package Concurrent_Map_Generic

Generic package Concurrent_Map_Generic provides a means of mapping values of one type into values of another type in multitasking programs. This translation from one type (called the *domain type*) to another type (called the *range type*) is a one-to-one mapping.

The package supports any number of concurrent read operations, but it serializes all update operations. Specifically, any number of Find/Eval/Is_Empty/Copy operations be done safely in parallel with a Define/Undefine/Make_Empty operation. Multiple Define/Undefine/Make_Empty operations are serialized. Iterators over the elements in a map can be used asynchronously; however, the sequence of values yielded may reflect the changes that have occurred in the map while the values are being accessed.

Several operations can perform checks for changing domain values that are already defined. A parameter selects whether this check allows the redefinition to occur or raises the Multiply_Defined exception (in this package). If any operation is passed an illegal input, the Constraint_Error exception is raised.

The implementation of this generic package uses a hash table. The size of the hash table (the number of buckets) is one of the formal parameters of the generic. The hash function, which spreads the values of the domain over the integer range, is also a parameter to the generic.

The formal parameters to the generic are:

```
generic
    Size : Integer;
    type Domain_Type is private;
    type Range_Type  is private;
    with function Hash (Key : Domain_Type) return Integer is <>;
package Concurrent_Map_Generic  is
    ...
end Concurrent_Map_Generic;
```

These parameters define the number of buckets in the hash table, the type of the domain, the type of the range, and the hash function.

function Cardinality

```
function Cardinality (The_Map : Map) return Natural;
```

Description

Returns the number of mappings defined in the specified map.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Natural;
```

Returns the number of mappings defined in the map.

Errors

This function raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

procedure Copy

```
procedure Copy (Target : in out Map;
                Source :        Map);
```

Description

Copies the contents of the source map into the target map.

This procedure first deletes all entries from the target and then adds an entry in the target for each entry in the source.

Parameters

```
Target : in out Map;
```

Specifies the map into which entries are to be copied.

```
Source : Map;
```

Specifies the map from which entries are to be copied.

Errors

This procedure raises the Constraint_Error exception when the value of either map is an uninitialized map.

procedure Define

```
procedure Define (The_Map        : in out Map;
                  D              :        Domain_Type;
                  R              :        Range_Type;
                  Trap_Multiples :        Boolean      := False);
```

Description

Defines a correspondence between a domain value and a range value.

This procedure creates entries in the map. The procedure traps multiple definitions of domain values when the Trap_Multiples parameter is true. When the parameter is false, the domain value is redefined to the new range value.

Parameters

`The_Map : in out Map;`

Specifies the map to which the entry is to be added.

`D : Domain_Type;`

Specifies the domain value to be added to the map.

`R : Range_Type;`

Specifies the range value that corresponds to the domain value.

`Trap_Multiples : Boolean := False;`

Specifies whether to trap an attempt to create a multiple definition of a domain value. When the parameter is true and a domain value that already exists in the map is given, the Multiply_Defined exception is raised. The default is not to trap multiple definitions.

Errors

This procedure raises the Multiply_Defined exception (in this package) when the value of the Trap_Multiples parameter is true and a domain value that already exists in the map is given.

This procedure also raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

generic formal type Domain_Type

```
type Domain_Type is private;
```

Description

Defines the type for the domain of the map.

The type must be a pure value type, and it must be constrained. The implicit operations of assignment and equality must work with values of this type.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Determines whether the iteration over the map is complete.

Parameters

```
Iter :  Iterator;
```

Specifies the iteration 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 (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

References

procedure Init

procedure Next

function Value

function Eval

```
function Eval (The_Map  : Map;
               D        : Domain_Type) return Range_Type;
```

Description

Returns a range value that corresponds to the specified domain value in the specified map.

When the given value of the domain does not exist, the Undefined exception is raised.

Parameters

`The_Map : Map;`

Specifies the map in which the domain value is to be found.

`D : Domain_Type;`

Specifies the value of the domain to be found in the map.

`return Range_Type;`

Returns the corresponding value of the range for the given domain value.

Errors

This function raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

The function also raises the Undefined exception (in this package) when the value of the domain type does not exist in the map.

procedure Find

```
procedure Find (The_Map :        Map;
                D       :        Domain_Type;
                R       : in out Range_Type;
                Success :    out Boolean);

procedure Find (The_Map :        Map;
                D       :        Domain_Type;
                P       : in out Pair;
                Success :    out Boolean);
```

Description

Finds the range value corresponding to the specified domain value (D) in the specified map.

This procedure finds the value of the range that corresponds to the domain value in the map. When the given value of the domain exists in the map, the R (range), or P (pair), parameter is updated with the domain/range values, and the Success parameter is returned true. When the given value of the domain does not exist, the R, or P, parameter is not changed, and the Success parameter is returned false.

Parameters

`The_Map : Map;`

Specifies the map in which the domain value is to be found.

`D : Domain_Type;`

Specifies the value of the domain to be found in the map.

`R : in out Range_Type;`

Returns the value of the range when the domain value is found. The parameter is not changed when the domain value is not found.

`P : in out Pair;`

Returns the domain/range pair when the domain value is found. The parameter is not changed when the domain value is not found.

`Success : out Boolean;`

Returns the value true when the domain value exists in the map; otherwise, the procedure returns false.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

---

generic formal function Hash

```
with function Hash (Key : Domain_Type) return Integer is <>;
```

Description

Defines the hash function used in the map.

This function takes a value of the domain type and creates an integer that is used to select a bucket in the hash table. That bucket will contain a corresponding value of the range type, if it exists. The integer returned by this function is normalized internally by the map operations to select a bucket.

In general, the map will perform better when the range of the integer returned by this function is much larger than the size of the hash table.

Parameters

```
Key : Domain_Type;
```

Specifies the domain value to be hashed.

```
return Integer is <>;
```

Returns the value of the hash function.

procedure Init

```
procedure Init (Iter     : out Iterator;
                The_Map  :     Map);
```

Description

Initializes the iterator to iterate through all the entries in the specified map.

When one or more entries exist in the map, the Value function returns the first entry in the map using this value of the iterator. When no entries exist in the map, the Done function returns the value true using this value of the iterator.

Parameters

`Iter : out Iterator;`

Returns the iterator.

`The_Map : Map;`

Specifies the map for which the iterator is desired.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

Example

This example demonstrates use of the iteration capability:

```
Init (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

References

function Done

procedure Next

function Value

---

procedure Initialize

```
procedure Initialize (The_Map : out Map);
```

Description

Creates an initialized and empty map.

This procedure creates a map; the map does not have to be created by the Nil function. The map contains no entries.

Parameters

```
The_Map :  out Map;
```

Specifies the value of the map to be created.

Errors

This procedure raises the Storage_Error exception when the map allocated by this procedure exceeds the allowed storage area.

References

function Is_Nil

function Nil

function Is_Empty

```
function Is_Empty (The_Map : Map) return Boolean;
```

Description

Determines whether the specified map is empty.

This function checks whether the map contains no entries. When a map is initialized or after the Make_Empty procedure is executed on the map, this condition is true.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Boolean;
```

Returns the value true when no entries exist in the map; otherwise, the function returns false.

Errors

This function raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

References

procedure Make_Empty

function Is_Nil

```
function Is_Nil (The_Map : Map) return Boolean;
```

Description

Returns true if the indicated map is null—that is, one that has not been initialized.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Boolean;
```

Returns the value true when the map is null; otherwise, the function returns false.

References

procedure Initialize

function Nil

---

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all entries in a map.

Objects of the Iterator type contain all of the information necessary to step over all of the entries in a map. The type is used with the Init and Next procedures and the Value and Done functions.

procedure Make_Empty

```
procedure Make_Empty (The_Map : in out Map);
```

Description

Clears the map of all entries.

Parameters

```
The_Map :  in out Map;
```

Specifies the map to be cleared.

Errors

This procedure raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

type Map

```
type Map is private;
```

Description

Defines the representation of the map.

Several important properties of the map type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the map are not copied but a new alias (a new name) for the map is created. Equality for this type has the property that the values of the maps are not compared but the names are compared. In other words, the operation of equality checks to determine whether any two map values designate the same map.

exception Multiply_Defined

```
Multiply_Defined : exception;
```

Description

Defines the exception raised by the Define procedure when the Domain parameter is already defined in the specified map and the value of the Trap_Multiples parameter is true.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Steps the iterator to point to the next entry in the map.

This procedure changes the iterator to point to the next entry in the map. When the iterator steps past the last entry, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
Init (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

References

function Done

procedure Init

function Value

function Nil

```
function Nil return Map;
```

Description

Returns a null map—that is, a map that is not initialized.

This function creates an uninitialized map. The map must be initialized before it can be used.

Parameters

```
return Map;
```

Returns a null map.

References

procedure Initialize

function Is_Nil

type Pair

```
type Pair is
    record
        D : Domain_Type;
        R : Range_Type;
    end record;
```

Description

Defines a record that contains a value of both the domain and the range.

generic formal type Range_Type

```
type Range_Type is private;
```

Description

Defines the type for the range of the map.

The type must be a pure value type, and it must be constrained. The implicit operations of assignment and equality must work with values of this type.

generic formal object Size

```
Size : Integer;
```

Description

Defines the size of the hash table in the map.

The value provided for this generic parameter depends on the hash function used and the number of expected or allowed values of the domain type. Preferred values are prime numbers.

procedure Undefine

```
procedure Undefine (The_Map : in out Map;
                    D       :        Domain_Type);
```

Description

Removes a map entry or entries for the domain value.

This procedure removes all entries for the specified domain value from the map. When the domain value does not exist in the map, the procedure raises the Undefined exception.

Parameters

`The_Map : in out Map;`

Specifies the map from which the entry is to be removed.

`D : Domain_Type;`

Specifies the domain value to be removed.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

The procedure also raises the Undefined exception (in this package) when the specified domain value does not already exist in the map.

exception Undefined

```
Undefined : exception;
```

Description

Defines the exception raised by the Eval function or the Undefine procedure when the Domain parameter does not exist in the specified map.

function Value

```
function Value (Iter : Iterator) return Domain_Type;
```

Description

Returns the domain value of the current entry pointed to by the iterator.

Parameters

```
Iter :  Iterator;
```

Specifies the iteration from which the domain is to be returned.

```
return Domain_Type;
```

Returns the domain value.

Example

This example demonstrates use of the iteration capability:

```
    Init (Device_Iterator, Device_Map);
    while not Done (Device_Iterator) loop
        ...
        ... Value (Device_Iterator);
        ...
        Next (Device_Iterator);
    end loop;
```

References

function Done

procedure Init

procedure Next

end Concurrent_Map_Generic;

RATIONAL

package Hash

This package defines simple hash functions returning Integer and Long_Integer values for Long_Integer and Ptr (pointer) types. These functions provide a many-to-one mapping between the input values and the output values. All functions are guaranteed not to raise any exceptions.

function Long_Integer_To_Integer

```
function Long_Integer_To_Integer (Value : Long_Integer) return Integer;
```

Description

Returns a hash value of the Integer type based on the value of the Value parameter.

This function provides a many-to-one mapping between long integers and integers.

Parameters

```
Value : Long_Integer;
```

Specifies the input value to be hashed.

```
return Integer;
```

Returns a many-to-one mapping from input values.

---

generic function Pointer_To_Integer

```
generic
    type T   is limited private;
    type Ptr is access T;
function Pointer_To_Integer (P : Ptr) return Integer;
```

function Pointer_To_Integer

```
function Pointer_To_Integer (P : Ptr) return Integer;
```

Description

Provides a many-to-one mapping between pointers and integers.

Parameters

```
P : Ptr;
```

Specifies the input value to be hashed.

```
return Integer;
```

Returns a many-to-one mapping from the input value.

generic formal type Ptr

```
type Ptr is access T;
```

Description

Defines the access type to be hashed.

generic formal type T

```
type T is limited private;
```

Description

Defines the type of object designated by the pointer to be hashed.

end Pointer_To_Integer;

generic function Pointer_To_Long_Integer

```
generic
    type T   is limited private;
    type Ptr is access T;
function Pointer_To_Long_Integer  (P : Ptr) return Long_Integer;
```

function Pointer_To_Long_Integer

```
function Pointer_To_Long_Integer (P : Ptr) return Long_Integer;
```

Description

Provides a many-to-one mapping between pointers and long integers.

Parameters

`P : Ptr;`

Specifies the input value to be hashed.

`return Long_Integer;`

Returns a many-to-one mapping from the input value.

---

generic formal type Ptr

```
type Ptr is access T;
```

Description

Defines the access type to be hashed.

generic formal type T

```
type T is limited private;
```

Description

Defines the type of object designated by the pointer to be hashed.

end Pointer_To_Long_Integer;

end Hash;

RATIONAL

generic package List_Generic

Generic package List_Generic provides a means of creating and manipulating abstract lists of elements. This generic allows lists of arbitrary size. There are operations for creating and adding items to lists, traversing and manipulating lists, and iterating over the items in lists.

If illegal values are provided to any of the operations in this package, the Constraint_Error exception is raised.

The formal parameter to the generic is:

```
generic
    type Element is private;
package List_Generic is
    ...
end List_Generic;
```

This parameter defines the kinds of elements that will make up lists.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Determines whether the iterator has cycled through all of the elements in a list.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator to be tested.

```
return Boolean;
```

Returns the value true when the iterator has stepped past the last element in the list; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_List);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

References

procedure Init

procedure Next

function Value

generic formal type Element

```
type Element is private;
```

Description

Defines the type of elements in lists.

The actual supplied for this type cannot be unconstrained.

function First

```
function First (L : List) return Element;
```

Description

Returns the first element in the list.

Parameters

```
L : List;
```

Specifies the list containing the desired element.

```
return Element;
```

Returns the first element of the list.

Errors

This function raises the Constraint_Error exception if the list is empty.

procedure Free

```
procedure Free (L : in out List);
```

Description

Reclaims the storage associated with the list and sets it to the empty list.

Parameters

```
L : in out List;
```

Specifies the list to be reclaimed.

procedure Init

```
procedure Init (Iter : out Iterator;
                L    : List);
```

Description

Initializes the iterator to iterate over the elements in the specified list.

When one or more elements exist in the list, the Value function returns the first element in the list using this value of the iterator. Successive values can be accessed by advancing the iterator to the next value using the Next procedure. When no elements exist in the list, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter : out Iterator;
```

Returns the iterator.

```
L : List;
```

Specifies the list of elements for which the iterator is desired.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_List);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

function Done

procedure Next

function Value

---

function Is_Empty

```
function Is_Empty (L : List) return Boolean;
```

Description

Determines whether the list is empty.

A list is empty if it contains no elements. This condition is true when a list is declared, when a list is set to the value returned by the Nil function, or after the Free procedure has been called with it.

Parameters

```
L : List;
```

Specifies the list to be checked.

```
return Boolean;
```

Returns true if the list is empty; otherwise, the function returns false.

References

procedure Free

function Nil

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all elements in a list.

Objects of this type can contain all of the information necessary to step over all of the elements in a list. The type is used with the Init and Next procedures and the Value and Done functions.

function Length

```
function Length (L : List) return Natural;
```

Description

Computes the number of elements in a list.

The length of an empty list is 0.

Parameters

```
L : List;
```

Specifies the list to be checked.

```
return Natural;
```

Returns the number of elements in the list. The function returns 0 if the list is empty.

type List

```
type List is private;
```

Description

Defines the representation of a list.

Several important properties of the List type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the list are not copied but a new alias (a new name) for the list is created. Equality for this type has the property that the values of the lists are not compared but the names are compared. In other words, the operation of equality checks to determine whether two list values designate the same list.

function Make

```
function Make (X : Element;
               L : List) return List;
```

Description

Adds the specified element to the front of the specified list and returns the new list.

Parameters

`X : Element;`

Specifies the element to be added.

`L : List;`

Specifies the list to which the element is to be added.

`return List;`

Returns the list that results from adding the element to the beginning of the initial list.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Advances the iterator to point to the next element in the list.

When the iterator steps past the last element, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be advanced.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_List);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

References

function Done

procedure Init

function Value

function Nil

```
function Nil return List;
```

Description

Returns the empty list containing no elements.

Parameters

```
return List;
```

Returns the empty list containing no elements.

function Rest

```
function Rest (L : List) return List;
```

Description

Returns the list of elements that remain after removing the first element from the list.

Parameters

```
L : List;
```

Specifies the list whose first element is to be removed.

```
return List;
```

Returns the list containing the elements of the initial list with the first element removed.

Errors

This function raises the Constraint_Error exception if the initial list is empty.

procedure Set_First

```
procedure Set_First (L     : List;
                     To_Be : Element);
```

Description

Replaces the first element in the list with the new element.

Parameters

`L : List;`

Specifies the list whose first element is to be replaced.

`To_Be :  Element;`

Specifies the new element.

Errors

This procedure raises the Constraint_Error exception if the initial list is empty.

procedure Set_Rest

```
procedure Set_Rest  (L     : List;
                     To_Be : List);
```

Description

Replaces all of the elements of the L parameter, other than its first element, with the elements in the list specified by the To_Be parameter.

This operation structurally modifies the list L by replacing the list of elements after its first element with the list To_Be. It does not create a copy of To_Be.

Parameters

`L : List;`

Specifies the list to be modified.

`To_Be :  List;`

Specifies the new value for the remaining elements in the list.

Errors

This procedure raises the Constraint_Error exception if the L parameter is empty.

function Value

```
function Value (Iter : Iterator) return Element;
```

Description

Returns the element pointed to by the iterator.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator from which the element is to be returned.

```
return Element;
```

Returns the element pointed to by the iterator.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_List);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

References

function Done

procedure Init

procedure Next

end List_Generic;

generic package Map_Generic

Generic package Map_Generic provides a means of mapping values of one type into values of another type. This translation from one type (called the ***domain type***) to another type (called the ***range type***) is a one-to-one mapping.

Several operations can perform checks for changing domain values that are already defined. A parameter selects whether this check allows the redefinition to occur or raises the Multiply_Defined exception (in this package). If any operation is passed an illegal input, the Constraint_Error exception is raised.

The implementation of this generic package uses a hash table. The size of the hash table (the number of buckets) is one of the formal parameters of the generic. The hash function, which spreads the values of the domain over the integer range, is also a parameter to the generic.

The formal parameters to the generic are:

```
generic
    Size : Integer;
    type Domain_Type is private;
    type Range_Type is private;
    with function Hash (Key : Domain_Type) return Integer is <>;
package Map_Generic is
    ...
end Map_Generic;
```

These parameters define the number of buckets in the hash table, the type of the domain, the type of the range, and the hash function.

function Cardinality

```
function Cardinality (The_Map : Map) return Natural;
```

Description

Returns the number of mappings defined in the specified map.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Natural;
```

Returns the number of mappings defined in the map.

Errors

This function raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

procedure Copy

```
procedure Copy (Target : in out Map;
                Source :        Map);
```

Description

Copies the contents of the source map into the target map.

This procedure first deletes all entries from the target and then adds an entry in the target for each entry in the source.

Parameters

```
Target : in out Map;
```

Specifies the map into which entries are to be copied.

```
Source : Map;
```

Specifies the map from which entries are to be copied.

Errors

This procedure raises the Constraint_Error exception when the value of either map is an uninitialized map.

procedure Define

```
procedure Define (The_Map        : in out Map;
                  D              :        Domain_Type;
                  R              :        Range_Type;
                  Trap_Multiples :        Boolean        := False);
```

Description

Defines a correspondence between a domain value and a range value.

This procedure creates entries in the map. The procedure traps multiple definitions of domain values when the Trap_Multiples parameter is true. When the parameter is false, the domain value is redefined to the new range value.

Parameters

`The_Map : in out Map;`

Specifies the map to which the entry is to be added.

`D : Domain_Type;`

Specifies the domain value to be added to the map.

`R : Range_Type;`

Specifies the range value that corresponds to the domain value.

`Trap_Multiples : Boolean := False;`

Specifies whether to trap an attempt to create a multiple definition of a domain value. When the parameter is true and a domain value that already exists in the map is given, the Multiply_Defined exception is raised. The default is not to trap multiple definitions.

Errors

This procedure raises the Multiply_Defined exception (in this package) when the value of the Trap_Multiples parameter is true and a domain value that already exists in the map is given.

The procedure also raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

generic formal type Domain_Type

```
type Domain_Type is private;
```

Description

Defines the type for the domain of the map.

The type must be a pure value type, and it must be constrained. The implicit operations of assignment and equality must work with values of this type.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Determines whether the iteration over the map is complete.

Parameters

```
Iter : Iterator;
```

Specifies the iteration 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 (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

References

procedure Init

procedure Next

function Value

function Eval

```
function Eval (The_Map  : Map;
               D        : Domain_Type) return Range_Type;
```

Description

Returns a range value that corresponds to the specified domain value in the specified map.

When the given value of the domain does not exist, the Undefined exception is raised.

Parameters

`The_Map : Map;`

Specifies the map in which to search for the domain value.

`D : Domain_Type;`

Specifies the value of the domain to find in the map.

`return Range_Type;`

Returns the corresponding value of the range for the given domain value.

Errors

This function raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

The function also raises the Undefined exception (in this package) when the value of the domain type does not exist in the map.

procedure Find

```
procedure Find (The_Map  :        Map;
                D        :        Domain_Type;
                R        : in out Range_Type;
                Success  :    out Boolean);

procedure Find (The_Map  :        Map;
                D        :        Domain_Type;
                P        : in out Pair;
                Success  :    out Boolean);
```

Description

Finds the range value that corresponds to the specified domain value in the specified map.

When the given value of the domain exists in the map, the R (range), or P (pair), parameter is updated with the domain/range values, and the Success parameter is returned true. When the given value of the domain does not exist, the R, or P, parameter is not changed, and the Success parameter is returned false.

Parameters

`The_Map : Map;`

Specifies the map in which to search for the domain value.

`D : Domain_Type;`

Specifies the value of the domain to find in the map.

`R : in out Range_Type;`

Returns the value of the range when the domain value is found. The parameter is not changed when the domain value is not found.

`P : in out Pair;`

Returns the domain/range pair when the domain value is found. The parameter is not changed when the domain value is not found.

`Success : out Boolean;`

Returns the value true when the domain value exists in the map; otherwise, the procedure returns false.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

---

generic formal function Hash

```
with function Hash (Key : Domain_Type) return Integer is <>;
```

Description

Defines the hash function used in the map.

This function takes a value of the domain type and creates an integer that is used to select a bucket in the hash table. That bucket will contain a corresponding value of the range type, if it exists. The integer returned by this function is normalized internally by the map operations to select a bucket.

In general, the map will perform better when the range of the integer returned by this function is much larger than the size of the hash table.

Parameters

```
Key : Domain_Type;
```

Specifies the domain value to be hashed.

```
return Integer is <>;
```

Returns the value of the hash function.

procedure Init

```
procedure Init (Iter    : out Iterator;
                The_Map :     Map);
```

Description

Initializes the iterator for the specified map.

This procedure initializes the iterator for the specified map. When one or more entries exist in the map, the Value function returns the first entry in the map using this value of the iterator. When no entries exist in the map, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter : out Iterator;
```

Returns the iterator.

```
The_Map : Map;
```

Specifies the map for which the iterator is desired.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

Example

This example demonstrates use of the iteration capability:

```
Init (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

procedure Init
package !Tools.Map_Generic

References

function Done

procedure Next

function Value

---

procedure Initialize

```
procedure Initialize (The_Map : out Map);
```

Description

Creates an initialized and empty map.

This procedure creates a map that contains no entries.

Parameters

```
The_Map : out Map;
```

Specifies the value of the map to be created.

Errors

This procedure raises the Storage_Error exception when the map allocated by this procedure exceeds the allowed storage area.

function Is_Empty

```
function Is_Empty (The_Map : Map) return Boolean;
```

Description

Determines whether the specified map is empty.

This function checks whether the map contains no entries. When a map is initialized or after the Make_Empty procedure is executed on the map, this condition is true.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Boolean;
```

Returns the value true when no entries exist in the map; otherwise, the function returns false.

Errors

This function raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

References

procedure Make_Empty

function Is_Nil

```
function Is_Nil (The_Map : Map) return Boolean;
```

Description

Returns true if the indicated map is null—that is, one that has not been initialized.

Parameters

```
The_Map : Map;
```

Specifies the map to be checked.

```
return Boolean;
```

Returns the value true when the map is null; otherwise, the function returns false.

References

procedure Initialize

function Nil

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all entries in a map.

Objects of the Iterator type contain all of the information necessary to step over all of the entries in a map. The Iterator type is used with the Init and Next procedures and the Value and Done functions.

procedure Make_Empty

```
procedure Make_Empty (The_Map : in out Map);
```

Description

Clears the map of all entries.

Parameters

```
The_Map :  in out Map;
```

Specifies the map to be cleared.

Errors

This procedure raises the Constraint_Error exception when The_Map parameter specifies an uninitialized map.

type Map

```
type Map is private;
```

Description

Defines the representation of the map.

Several important properties of the Map type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the map are not copied but a new alias (a new name) for the map is created. Equality for this type has the property that the values of the maps are not compared but the names are compared. In other words, the operation of equality checks to determine whether any two map values designate the same map.

exception Multiply_Defined

```
Multiply_Defined : exception;
```

Description

Defines the exception raised by the Define procedure when the Domain parameter is already defined in the specified map and the value of the Trap_Multiples parameter is true.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Steps the iterator to point to the next entry in the map.

This procedure changes the iterator to point to the next entry in the map. When the iterator steps past the last entry, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
Init (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

References

function Done

procedure Init

function Value

function Nil

```
function Nil return Map;
```

Description

Returns a null map—that is, a map that is not initialized.

This function creates an uninitialized map. The map must be initialized before it can be used.

Parameters

```
return Map;
```

Returns a null map.

References

procedure Initialize

function Is_Nil

type Pair

```
type Pair is
    record
        D : Domain_Type;
        R : Range_Type;
    end record;
```

Description

Defines a record that contains a value of both the domain and the range.

generic formal type Range_Type

```
type Range_Type is private;
```

Description

Defines the type for the range of the map.

The type must be a pure value type, and it must be constrained. The implicit operations of assignment and equality must work with values of this type.

generic formal object Size

```
Size : Integer;
```

Description

Defines the size of the hash table in the map.

The value provided for this generic parameter depends on the hash function used and the number of expected or allowed values of the domain type. Preferred values are prime numbers.

procedure Undefine

```
procedure Undefine (The_Map : in out Map;
                    D       :        Domain_Type);
```

Description

Removes a map entry or entries for the domain value.

This procedure removes all entries for the specified domain value from the map. When the domain value does not exist in the map, the procedure raises the Undefined exception.

Parameters

```
The_Map :  in out Map;
```

Specifies the map from which the entry is to be removed.

```
D : Domain_Type;
```

Specifies the domain value to be removed.

Errors

This procedure raises the Constraint_Error exception when the value of The_Map parameter is an uninitialized map.

The procedure also raises the Undefined exception (in this package) when the specified domain value does not already exist in the map.

exception Undefined

```
Undefined : exception;
```

Description

Defines the exception raised by the Eval function or the Undefine procedure when the Domain parameter does not exist in the specified map.

function Value

```
function Value (Iter : Iterator) return Domain_Type;
```

Description

Returns the domain value of the current entry pointed to by the iterator.

Parameters

`Iter : Iterator;`

Specifies the iteration from which the domain is to be returned.

`return Domain_Type;`

Returns the domain value.

Example

This example demonstrates use of the iteration capability:

```
Init (Device_Iterator, Device_Map);
while not Done (Device_Iterator) loop
    ...
    ... Value (Device_Iterator);
    ...
    Next (Device_Iterator);
end loop;
```

References

function Done

procedure Init

procedure Next

end Map_Generic;

generic package Queue_Generic

Generic package Queue_Generic provides a means of creating and manipulating abstract queues of elements. This generic allows queues of arbitrary size. There are operations for creating queues, adding and removing elements from them, and iterating over the items in them.

If illegal values are provided to any of the operations in this package, the Constraint_Error exception is raised.

The formal parameter to the generic is:

```
generic
    type Element is private;
package Queue_Generic is
    ...
end Queue_Generic;
```

This parameter defines the kinds of elements that will be queued.

procedure Add

```
procedure Add (Q : in out Queue;
               X :        Element);
```

Description

Adds an element to the end of the queue.

Parameters

`Q : in out Queue;`

Specifies the queue to which the element is to be added.

`X : Element;`

Specifies the element to be added.

procedure Copy

```
procedure Copy (Target : in out Queue;
                Source :        Queue);
```

Description

Deletes any elements in the target, initializes the target if necessary, and copies all of the elements in the source into the target.

Parameters

```
Target :  in out Queue;
```

Specifies the new queue created from the elements in the source.

```
Source :  Queue;
```

Specifies the source of the elements to be used in creating the new queue.

procedure Delete

```
procedure Delete (Q : in out Queue);
```

Description

Removes the item at the beginning of the queue from the queue.

The item at the beginning of the queue is the same as the item returned by the First function.

Parameters

```
Q : in out Queue;
```

Specifies the queue from which the first item is to be deleted.

Errors

This procedure raises the Constraint_Error exception if the queue is empty.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Determines whether the iterator has cycled through all of the elements in a queue.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator to be tested.

```
return Boolean;
```

Returns the value true when the iterator has stepped past the last element in the queue; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Queue);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

procedure Init

procedure Next

function Value

generic formal type Element

```
type Element is private;
```

Description

Defines the type of elements in queues.

The actual supplied for this type cannot be unconstrained.

function First

```
function First (Q : Queue) return Element;
```

Description

Returns the item at the beginning of the queue.

Parameters

```
Q : Queue;
```

Specifies the queue from which the item is to be retrieved.

```
return Element;
```

Returns the first item in the queue.

Errors

This function raises the Constraint_Error exception if the queue is empty.

procedure Init

```
procedure Init (Iter : out Iterator;
                Q    :     Queue);
```

Description

Initializes the iterator to iterate over the elements in the specified queue.

When one or more elements exist in the queue, the Value function returns the first element in the queue using this value of the iterator. Successive values can be accessed by advancing the iterator to the next value using the Next procedure. When no elements exist in the queue, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter : out Iterator;
```

Returns the iterator.

```
Q : Queue;
```

Specifies the queue of elements for which the iterator is desired.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Queue);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

function Done

procedure Next

function Value

---

procedure Initialize

```
procedure Initialize (Q : out Queue);
```

Description

Creates an empty queue.

Note that this procedure is provided for consistency between this package and other similar packages. However, it has no effect and need not be called because all the objects of the Queue type that are declared are implicitly initialized.Queue type; Queue_Generic.Queue; Initialize procedure

---

Parameters

`Q : out Queue;`

Specifies the empty queue.

---

function Is_Empty

```
function Is_Empty (Q : Queue) return Boolean;
```

Description

Determines whether the queue is empty.

Parameters

```
Q : Queue;
```

Specifies the queue to be checked.

```
return Boolean;
```

Returns true if the queue contains no elements; otherwise, the function returns false.

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all elements in a queue.

Objects of this type can contain all of the information necessary to step over all of the elements in a queue. The type is used with the Init and Next procedures and the Value and Done functions.

procedure Make_Empty

```
procedure Make_Empty (Q : in out Queue);
```

Description

Removes all of the elements from the queue, making it empty.

Parameters

```
Q : in out Queue;
```

Specifies the queue to be emptied.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Advances the iterator to point to the next element in the queue.

When the iterator steps past the last element, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be advanced.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Queue);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

function Done

procedure Init

function Value

type Queue

```
type Queue is private;
```

Description

Defines the representation of a queue.

Several important properties of the Queue type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the queue are not copied but a new alias (a new name) for the queue is created. Equality for this type has the property that the values of the queues are not compared but the names are compared. In other words, the operation of equality checks to determine whether two queue values designate the same list.

---

function Value

```
function Value (Iter : Iterator) return Element;
```

Description

Returns the element pointed to by the iterator.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator from which the element is to be returned.

```
return Element;
```

Returns the element pointed to by the iterator.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Queue);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

function Done

procedure Init

procedure Next

end Queue_Generic;

generic package Set_Generic

Generic package Set_Generic provides a means of creating and manipulating abstract sets of objects. This generic allows sets of arbitrary size. There are operations for adding members of the set, deleting members of the set, checking for membership in the set, and iterating over all members of the set.

If illegal values are passed to any of the operations in this package, the Constraint_Error exception is raised.

The formal parameter to the generic is:

```
generic
    type Element is private;
package Set_Generic is
    ...
end Set_Generic;
```

This parameter can be of any pure value type.

procedure Add

```
procedure Add (S : in out Set;
               X :        Element);
```

Description

Adds the specified element to the specified set.

When the element is already in the set, the procedure does not alter the set.

Parameters

`S : in out Set;`

Specifies the set to which the element is to be added.

`X : Element;`

Specifies the element to be added.

procedure Copy

```
procedure Copy (Target : in out Set;
                Source :        Set);
```

Description

Copies the contents of the source set into the target set.

This procedure first removes all elements from the target and then adds all elements in the source to the target.

Parameters

`Target : in out Set;`

Specifies the set into which elements are to be copied.

`Source : Set;`

Specifies the set from which elements are to be copied.

procedure Delete

```
procedure Delete (S : in out Set;
                  X :        Element);
```

Description

Deletes the specified element from the specified set.

When the element is not already in the set, the procedure does not alter the set.

Parameters

`S : in out Set;`

Specifies the set from which the element is to be deleted.

`X : Element;`

Specifies the element to be deleted.

function Done

```
function Done (Iter : Iterator) return Boolean;
```

Description

Determines whether the iteration over the set is complete.

This function checks whether the iterator has cycled through all of the elements in the set.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator to be tested.

```
return Boolean;
```

Returns the value true when the iterator has stepped past the last element in the set; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Set);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

procedure Init

procedure Next

function Value

generic formal type Element

```
type Element is private;
```

Description

Defines the type of elements in the set.

The type must be a pure value type. The implicit operations of assignment and equality must work with values of this type.

procedure Init

```
procedure Init (Iter : out Iterator;
                S    :     Set);
```

Description

Initializes the iterator for the specified set.

When one or more elements exist in the set, the Value function returns the first element in the set using this value of the iterator. When no elements exist in the set, the Done function returns the value true using this value of the iterator.

Parameters

`Iter : out Iterator;`

Returns the iterator.

`S : Set;`

Specifies the set for which the iterator is desired.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_Set);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

References

function Done

procedure Next

function Value

procedure Initialize

```
procedure Initialize (S : out Set);
```

Description

Creates an initialized and empty set.

This procedure creates a set that contains no elements. Objects of Set type are initially empty, so the use of this procedure is not required.

Parameters

```
S : out Set;
```

Specifies the set to be created.

function Is_Empty

```
function Is_Empty (S : Set) return Boolean;
```

Description

Determines whether the specified set is empty.

This function checks whether the set contains no elements. When a set is declared or initialized, or after the Make_Empty procedure is executed on the set, this condition is true.

Parameters

```
S : Set;
```

Specifies the set to be checked.

```
return Boolean;
```

Returns the value true when no elements exist in the set; otherwise, the function returns false.

References

procedure Make_Empty

function Is_Member

```
function Is_Member (S : Set;
                    X : Element) return Boolean;
```

Description

Determines whether the specified element is a member of the specified set.

Parameters

```
S : Set;
```

Specifies the set to be searched for the element.

```
X : Element;
```

Specifies the element for which to search.

```
return Boolean;
```

Returns the value true when the specified element is a member of the specified set; otherwise, the function returns false.

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all elements in a set.

Objects of this type contain all of the information necessary to step over all of the elements in a set. The type is used with the Init and Next procedures and the Value and Done functions.

procedure Make_Empty

```
procedure Make_Empty (S : in out Set);
```

Description

Clears the set of all elements.

Parameters

```
S : in out Set;
```

Specifies the set to be cleared.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Steps the iterator to point to the next element in the set.

This procedure changes the iterator to point to the next element in the set. Although the set is unordered, the iterator steps through all elements one by one. When the iterator steps past the last element, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be stepped.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Set);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

function Done

procedure Init

function Value

type Set

```
type Set is private;
```

Description

Defines the representation of the set.

Several important properties of the Set type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the set are not copied but a new alias (a new name) for the set is created. Equality for this type has the property that the values of the sets are not compared but the names are compared. In other words, the operation of equality checks to determine whether two set values designate the same set.

function Value

```
function Value (Iter : Iterator) return Element;
```

Description

Returns the element pointed to by the iterator.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator from which to return the element.

```
return Element;
```

Returns the element pointed to by the iterator.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_Set);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

References

function Done

procedure Init

procedure Next

end Set_Generic;

RATIONAL

package Simple_Status

This package provides an abstraction for simple error status reporting. It defines a type, the Condition type, that can be used to return error information from subprogram calls. The status returned from some Environment interfaces is of the Condition type and can be interrogated/manipulated with operations provided in this package. Users can also use this abstraction when implementing error reporting in their own applications.

A condition consists of a condition name and a message. The condition name indicates the type of error (if any), the severity of the error, and whether the operation completed successfully. The message provides additional information about the error.

By convention, condition names in an application should be standardized so that error conditions can be tested programmatically. In simple applications, a condition name alone can be used to indicate status.

Objects of the condition type are relatively large; where space and time considerations are important, they should be passed with ***in out*** mode so that copies are not made when subprograms are called.

Example

The following program illustrates the use of some of the facilities of package Simple_Status:

```
with Io;
with Log;
with Simple_Status;

procedure Simple_Status_Example is

    package Ss renames Simple_Status;

    Status : Ss.Condition;
    Name : Ss.Condition_Name;

begin
```

```
    Ss.Create_Condition (Status,
                         Error_Type => "Syntax error",
                         Severity => Ss.Problem,
                         Message => "occurred at this point in processing");

    Log.Put_Condition (Status);
    Io.Put_Line (Ss.Display_Message (Status));
    Io.New_Line;

    Io.Put_Line ("Error type: " & Ss.Name (Status));
    Io.Put_Line ("Severity: " & Ss.Condition_Class'Image
                                  (Ss.Severity (Status)));
    Io.Put_Line ("Message: " & Ss.Message (Status));
    Io.New_Line;

    if Ss.Error (Status) then
        Io.Put_Line ("It was an error");
    end if;

end Simple_Status_Example;
```

If this program is executed, it generates the following output:

```
87/01/17 19:18:28 ***  Syntax error occurred at this point in processing.
Syntax error occurred at this point in processing

Error type: Syntax error
Severity: PROBLEM
Message: occurred at this point in processing

It was an error
```

type Condition

```
type Condition is private;
```

Description

Defines a status condition that can be used to return error information from subprogram calls.

A condition consists of a condition name and a message. The condition name indicates the type of error (if any), the severity of the error, and whether the operation completed successfully. The message provides additional information about the error.

Conditions are self-initializing—objects of the Condition type will have null strings for the error type and message components, and they will have normal severity. They can also be initialized by calling the Initialize procedure. The condition that results from calling Initialize represents successful completion.

Objects of the Condition type are relatively large; where space and time considerations are important, they should be passed with *in out* mode so that copies are not made when subprograms are called.

References

procedure Initialize

type Condition_Class

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

Description

Defines the class of error condition (if any) that resulted from an operation.

The class of an error condition is part of its condition name.

Enumerations

`Fatal`
Indicates that the operation did not complete and proceeding is dangerous.

`Normal`
Indicates that the operation completed normally.

`Problem`
Indicates that the operation did not complete and it is safe to proceed.

`Warning`
Indicates that the operation completed and it is safe to proceed, but something unexpected happened.

type Condition_Name

```
type Condition_Name is private;
```

Description

Defines the name of an error condition.

The name of a condition consists of its type (a string limited to 63 characters) and its severity level (of Condition_Class type).

procedure Create_Condition

```
procedure Create_Condition (Status     : in out Condition;
                            Error_Type :        String;
                            Message    :        String          := "";
                            Severity   :        Condition_Class := Problem);

procedure Create_Condition (Status     : in out Condition;
                            Error_Type :        Condition_Name;
                            Message    :        String          := "");
```

Description

Creates a condition of the indicated type and severity, with the message supplied, and returns it in the Status parameter.

Parameters

`Status : in out Condition;`

Returns the newly created condition.

`Error_Type : String;`

`Error_Type : Condition_Name;`

Specifies the name and/or type of condition to be created. The type of condition must be a string of 63 or fewer characters. If longer, the string is truncated.

`Message : String := "";`

Specifies the message of the condition to be created. The default is to create a condition with an empty message.

`Severity : Condition_Class := Problem;`

Specifies the severity of the condition to be created. The default is to create a condition with problem severity.

function Create_Condition_Name

```
function Create_Condition_Name (Error_Type : String;
                                Severity   : Condition_Class := Problem)
                                              return Condition_Name;
```

Description

Creates a condition name of the indicated type and severity.

Parameters

```
Error_Type : String;
```

Specifies the type of the condition name to be created. The type of condition must be a string of 63 or fewer characters. If longer, the string is truncated.

```
Severity : Condition_Class := Problem;
```

Specifies the severity of the condition name to be created. The default is to create a condition with problem severity.

```
return Condition_Name;
```

Returns the condition name created.

function Display_Message

```
function Display_Message (Status : Condition) return String;
```

Description

Returns the condition name and message of the indicated condition.

Typically, this function is used when composing error messages for presentation to users. The string that is returned is text that describes the result of an operation and the causes of errors.

Note that the Log.Put_Condition procedure (SJM) can be used to display the same information in logging format to the Current_Output window or file.

Parameters

```
Status : Condition;
```

Specifies the condition to interrogate.

```
return String;
```

Returns the condition name (in string form) and the message of the condition.

References

SJM, procedure Log.Put_Condition

function Equal

```
function Equal (Status     : Condition;
                Error_Type : String) return Boolean;

function Equal (Status     : Condition;
                Error_Type : Condition_Name) return Boolean;

function Equal (Status     : Condition_Name;
                Error_Type : String) return Boolean;

function Equal (Status     : Condition_Name;
                Error_Type : Condition_Name) return Boolean;
```

Description

Determines whether the type of error of the condition supplied by the Status parameter is the same as the error type supplied by the Error_Type parameter.

The comparison is based on a case-sensitive character string comparison. The severity level does not participate in the comparison.

Parameters

`Status : Condition;`

`Status : Condition_Name;`

Specifies the condition for which to check the error type.

`Error_Type : String;`

`Error_Type : Condition_Name;`

Specifies the error type for which to check.

`return Boolean;`

Returns true if the type of error of the condition supplied by the Status parameter is the same as the error type supplied by the Error_Type parameter; otherwise, the function returns false. The comparison is based on a case-sensitive character string comparison. The severity level does not participate in the comparison.

function Error

```
function Error (Error_Type : Condition_Name;
                Level      : Condition_Class := Warning) return Boolean;

function Error (Status : Condition;
                Level  : Condition_Class := Warning) return Boolean;
```

Description

Determines whether the severity level of the indicated condition is worse than the indicated level.

Typically, the Error function is used to determine whether an operation that returns a condition completed successfully. If the operation failed, a call to the Error function with the condition resulting from the failed operation will return the value true. If the operation did not complete successfully, the Display_Message function or the Log.Put_Condition procedure (SJM) can be used to determine the cause of the error.

The levels of severity are normal (lowest), warning, problem, and fatal (highest).

Parameters

`Error_Type : Condition_Name;`

`Status : Condition;`

Specifies the condition for which to check the severity.

`Level : Condition_Class := Warning;`

Specifies the level with which to compare. The default is to compare with a warning condition.

`return Boolean;`

Returns true if the severity of the condition is greater than or equal to the level supplied; otherwise, the function returns false. The levels of severity are normal (lowest), warning, problem, and fatal (highest).

References

function Display_Message

SJM, procedure Log.Put_Condition

---

function Error_Type

```
function Error_Type (Status : Condition) return Condition_Name;
```

Description

Returns the name of the indicated condition.

Parameters

```
Status : Condition;
```

Specifies the condition to be interrogated.

```
return Condition_Name;
```

Returns the name of the condition.

procedure Initialize

```
procedure Initialize (Status : in out Condition);
```

Description

Sets the severity level of the indicated condition to normal.

The normal value represents a successful completion. The condition name and message fields will be set to the null string.

Parameters

```
Status :  in out Condition;
```

Returns the condition with the severity level set to normal. The procedure also sets the condition name and message fields to the null string.

function Message

```
function Message (Status : Condition) return String;
```

Description

Returns the message of the indicated condition.

Parameters

```
Status :  Condition;
```

Specifies the condition to be interrogated.

```
return String;
```

Returns the message of the indicated condition.

function Name

```
function Name (Error_Type : Condition_Name) return String;

function Name (Status : Condition) return String;
```

Description

Returns the type of error condition for the indicated condition.

Parameters

```
Error_Type : Condition_Name;

Status : Condition;
```

Specifies the condition to be interrogated.

```
return String;
```

Returns the type of error condition. **Note that this string will be less than or equal** to 63 characters long.

function Severity

```
function Severity (Error_Type : Condition_Name) return Condition_Class;

function Severity (Status : Condition) return Condition_Class;
```

Description

Returns the severity of the indicated condition.

Parameters

```
Error_Type : Condition_Name;

Status : Condition;
```

Specifies the condition to be interrogated.

```
return Condition_Class;
```

Returns the severity of the error condition.

end Simple_Status;

generic package Stack_Generic

Generic package Stack_Generic provides a means of creating and manipulating abstract stacks of elements. This generic allows stacks of arbitrary size. There are operations for creating stacks, pushing and popping elements on and off stacks, and iterating over the elements in stacks.

If an attempt is made to pop or read an element off an empty stack, the Underflow exception (in this package) is raised.

The formal parameter to the generic is:

```
generic
    type Element is private;
package Stack_Generic is
    ...
end Stack_Generic;
```

This parameter defines the kinds of elements that are kept in stacks.

procedure Copy

```
procedure Copy (Target : in out Stack;
                Source :        Stack);
```

Description

Removes any elements in the target, initializing it if necessary, and then copies the elements in the source into it.

Parameters

```
Target : in out Stack;
```

Specifies the new stack.

```
Source : Stack;
```

Specifies the stack to be copied.

function Done

```
function Done  (Iter : Iterator) return Boolean;
```

Description

Determines whether the iterator has cycled through all of the elements in a stack.

Parameters

```
Iter :  Iterator;
```

Specifies the iterator to be tested.

```
return Boolean;
```

Returns the value true when the iterator has stepped past the last element in the stack; otherwise, the function returns false.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_Stack);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

References

procedure Init

procedure Next

function Value

generic formal type Element

```
type Element is private;
```

Description

Defines the type of elements in stacks.

The actual supplied for this type cannot be unconstrained.

function Empty

```
function Empty (S : Stack) return Boolean;
```

Description

Determines whether there are any elements in the stack.

Parameters

```
S : Stack;
```

Specifies the stack to be queried.

```
return Boolean;
```

Returns the value true if there are any elements in the specified stack; otherwise, the function returns false.

constant Empty_Stack

```
Empty_Stack : constant Stack;
```

Description

Defines an empty stack containing no elements.

procedure Init

```
procedure Init (Iter : out Iterator;
                S    :     Stack);
```

Description

Initializes the iterator to iterate over the elements in the specified stack.

When one or more elements exist in the stack, the Value function returns the first element in the list using this value of the iterator. Successive values can be accessed by advancing the iterator to the next value using the Next procedure. When no elements exist in the stack, the Done function returns the value true using this value of the iterator.

Parameters

```
Iter : out Iterator;
```

Returns the iterator.

```
S : Stack;
```

Specifies the stack of elements for which the iterator is desired.

Example

This example demonstrates use of the iteration capability:

```
    Init (Sensor_Iterator, Sensor_Stack);
    while not Done (Sensor_Iterator) loop
        ...
        ... Value (Sensor_Iterator);
        ...
        Next (Sensor_Iterator);
    end loop;
```

procedure Init
package !Tools.Stack_Generic

References

function Done

procedure Next

function Value

---

type Iterator

```
type Iterator is private;
```

Description

Defines a type that allows iterating over all elements in a list.

Objects of this type can contain all of the information necessary to step over all of the elements in a list. The type is used with the Init and Next procedures and the Value and Done functions.

procedure Make_Empty

```
procedure Make_Empty (S : in out Stack);
```

Description

Removes all of the elements in the stack and sets it to the value of the Empty_Stack constant.

Parameters

```
S : in out Stack;
```

Specifies the stack to be emptied.

procedure Next

```
procedure Next (Iter : in out Iterator);
```

Description

Advances the iterator to point to the next element in the stack.

When the iterator steps past the last element, the Done function returns the value true.

Parameters

```
Iter :  in out Iterator;
```

Specifies the iterator to be advanced.

Errors

This procedure raises the Constraint_Error exception if the iterator is uninitialized or has no elements in it.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Stack);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

References

function Done

procedure Init

function Value

procedure Pop

```
procedure Pop (S : in out Stack);
```

Description

Removes the last item pushed onto the specified stack.

Parameters

```
S : in out Stack;
```

Specifies the stack from which the item is to be removed.

Errors

This procedure raises the Underflow exception (in this package) if there are no elements in the stack.

procedure Push

```
procedure Push (X :        Element;
                S : in out Stack);
```

Description

Pushes the specified element onto the top of the specified stack.

Parameters

```
X : Element;
```

Specifies the element to be placed on the stack.

```
S : in out Stack;
```

Specifies the stack on which the element is to be placed.

type Stack

```
type Stack is private;
```

Description

Defines the representation of a stack.

Several important properties of the Stack type are visible through the implicit operations of assignment and equality. Assignment for this type has the property that the contents of the stack are not copied but a new alias (a new name) for the stack is created. Equality for this type has the property that the values of the stacks are not compared but the names are compared. In other words, the operation of equality checks to determine whether two stack values designate the same stack.

---

function Top

```
function Top (S : Stack) return Element;
```

Description

Returns the last element pushed onto the specified stack.

Parameters

```
S : Stack;
```

Specifies the stack from which the element is to be retrieved.

```
return Element;
```

Returns the last element pushed onto the stack.

Errors

This function raises the Underflow exception (in this package) if the stack is empty.

exception Underflow

```
Underflow : exception;
```

Description

Defines an exception raised when an attempt is made to pop or read elements from empty stacks.

function Value

```
function Value (Iter : Iterator) return Element;
```

Description

Returns the element pointed to by the iterator.

Parameters

```
Iter : Iterator;
```

Specifies the iterator from which the element is to be returned.

```
return Element;
```

Returns the element pointed to by the iterator.

Errors

This function raises the Constraint_Error exception if the iterator is uninitialized or has no more elements in it.

Example

This example demonstrates use of the iteration capability:

```
Init (Sensor_Iterator, Sensor_Stack);
while not Done (Sensor_Iterator) loop
    ...
    ... Value (Sensor_Iterator);
    ...
    Next (Sensor_Iterator);
end loop;
```

function Value
package !Tools.Stack_Generic

References

function Done

procedure Init

procedure Next

---

end Stack_Generic;

package Standard

```
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)
                    -- -3.0517578125000E-05
                  range -(2.0**32) .. (2.0**32) - (2.0**(-15));
                    -- -4.2949672960000E+09 .. 4.2949672960000E+09

    Constraint_Error  : exception;
    Numeric_Error     : exception;
    Program_Error     : exception;
    Storage_Error     : exception;
    Tasking_Error     : exception;
end Standard;
```

Description

Ada requires a package called Standard that defines all predefined identifiers in the language. The *Reference Manual for the Ada Programming Language* contains its general description in Section 8.6, "The Package Standard." The specification for package Standard for the Rational architecture is given in this section.

package System

```
package System is
    type Address is private;
    Null_Address : constant Address;
    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;
```

Description

Ada requires a predefined library package called System that includes the definitions of certain configuration-dependent characteristics. The *Reference Manual for the Ada Programming Language* contains the general specification in Section 13.7, "The Package System." The Rational Environment implementation specification of package System is given in this section.

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.

This package System is for the R1000. Other targets have their own package System documented with their target-specific information.

```
type Address is private;
```

Defines the type returned by the predefined attribute 'Address.

```
Assertion_Error : exception;
```

Defines the exception raised by the Unchecked_Conversion function when the object resulting from the conversion has bounds bigger than the Target type allows.

```
Bit : constant := 1;
```

Defines a constant that represents the size of a single bit.

```
type Byte is new Natural range 0 .. 255;
```

Defines the representation for a byte of data.

```
Byte_Size : constant := 8 * Bit;
```

Defines the size of a byte.

```
type Byte_String is array (Natural range <>) of Byte;
```

Defines a type that represents a string of bytes. This type is used in some I/O packages to represent the data going to or from terminals or tapes.

```
Capability_Error : exception;
```

Defines the exception raised by the Unchecked_Conversion function when the conversion fails.

```
Fine_Delta : constant := 1.0 / (2.0 ** 63);
```

Defines the smallest delta allowed in a fixed-point constraint that has the range constraint −1.0 .. 1.0.

This constant is of Universal_Real type.

```
Max_Digits : constant := 15;
```

Defines the largest value allowed for the number of significant decimal digits in a floating-point constraint.

This constant is of Universal_Integer type.

```
Max_Int : constant := Long_Integer'Pos (Long_Integer'Last);
```

Defines the largest (most positive) value of all predefined integer types.

This constant is of Universal_Integer type.

```
Max_Mantissa : constant := 63;
```

Defines the largest possible number of binary digits in the mantissa of model numbers of a fixed-point subtype.

This constant is of Universal_Integer type.

```
Megabyte : constant := (2 ** 20) * Byte_Size;
```

Defines a constant for the number 1,048,576.

This constant is of Universal_Integer type.

```
Memory_Size : constant := 32 * Megabyte;
```

Defines the number of available storage units in the configuration.

This constant is of Universal_Integer type.

```
Min_Int : constant := Long_Integer'Pos (Long_Integer'First);
```

Defines the smallest (most negative) value of all predefined integer types.

This constant is of Universal_Integer type.

```
type Name is (R1000);
```

Defines values of alternative machine configurations handled by the Environment.

```
Null_Address : constant Address;
```

Defines a null value of type Address.

```
subtype Priority is Integer range 0 .. 5;
```

Defines the range of task priorities available for a task within a job.

Task priorities are assigned according to the rules of the *Reference Manual for the Ada Programming Language.* These priorities differ from job priorities, which are managed by procedures in SJM, package Job.

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

Defines the number of bits per storage unit.

This constant is of Universal_Integer type.

```
System_Name : constant Name := R1000;
```

Defines the value of the default system name.

```
Tick : constant := 200.0E-9;
```

Defines the basic clock period, in seconds (that is, 200 nanoseconds).

This constant is of Universal_Real type.

```
Type_Error : exception;
```

Defines the exception raised by the Unchecked_Conversion function when the underlying architectural types used to represent the Source and Target types are not the same, even though the types are compatible in an Ada sense.

```
Word_Size : constant := 128 * Bit;
```

Defines the size of addressed words in the system.

---

generic procedure Table_Sort_Generic

The Table_Sort_Generic generic procedure provides a table sorting capability. The formal parameters to the generic include the table of elements to be sorted, the element type, the size of the table, and a comparison function that defines the ordering of the elements in the table. The generic procedure takes an unsorted table of the size and type defined by those parameters and returns the sorted table.

The formal parameters to the generic are:

```
generic
   type Element is private;
   type Index is (<>);
   type Element_Array is array (Index range <>) of Element;
   with function "<" (Left  : Element;
                      Right : Element) return Boolean is <>;
procedure Table_Sort_Generic (Table : in out Element_Array);
```

generic formal function "<"

```
with function "<" (Left  : Element;
                   Right : Element) return Boolean is <>;
```

Description

Defines the function that is to be used to order the elements in the table.

The comparison must meet the following condition: if A is less than B, then B is not less than A. This ensures that if A and B are transposed during a pass of the sort, they won't be transposed again in a subsequent pass.

Parameters

`Left : Element;`

Specifies the left element in the comparison.

`Right : Element;`

Specifies the right element in the comparison.

`return Boolean is <>;`

Returns the value true when the left element is less than the right element; otherwise, the function returns false.

---

generic formal type Element

```
type Element is private;
```

Description

Defines the types of elements in the table.

generic formal type Element_Array

```
type Element_Array is array (Index range <>) of Element;
```

Description

Defines the type of table to be sorted.

generic formal type Index

```
type Index is (<>);
```

Description

Defines the index of the table.

---

procedure Table_Sort_Generic

```
procedure Table_Sort_Generic (Table : in out Element_Array);
```

Description

Sorts the specified table according to the actual parameters of the generic.

This generic procedure sorts the table. The sorting algorithm is defined by the generic actual parameters to the generic. The internal method for sorting is a Shell sort.

Parameters

```
Table :  in out Element_Array;
```

Specifies the table to be sorted.

end Table_Sort_Generic;

package Time_Utilities

Package Time_Utilities contains a number of utilities for manipulating times and dates. Package Calendar contains one representation for time and subprograms for manipulating time. Package Time_Utilities contains two important type definitions for alternative representations for time and subprograms for manipulating these alternatives. These are Time type and Interval type.

The package contains:

- Operations to convert from one representation to another.
- Image and value operations to convert time to strings and back.
- Operations to find the current time in these alternate representations.
- Constants and types that represent units of time such as minutes, hours, seconds, months, and years.

Unless otherwise specified, the Constraint_Error exception is raised if illegal values are passed to any of the operations in this package.

function "+"

```
function "+" (D : Weekday;
              I : Integer) return Weekday;
```

Description

Computes the weekday that it will be after the specified number of days have elapsed from the specified weekday.

Parameters

```
D : Weekday;
```

Specifies the weekday from which to start.

```
I : Integer;
```

Specifies the number of days that are to elapse.

```
return Weekday;
```

Returns the day that it will be after the indicated number of days have elapsed.

function "–"

```
function "-" (D : Weekday;
              I : Integer) return Weekday;
```

Description

Computes the weekday that it will be after the specified number of days are subtracted from the specified weekday.

Parameters

```
D : Weekday;
```

Specifies the weekday to start from.

```
I : Integer;
```

Specifies the number of days to be subtracted.

```
return Weekday;
```

Returns the day that it will be after the indicated number of days are subtracted.

function Convert

```
function Convert (I : Interval) return Duration;

function Convert (D : Duration) return Interval;
```

Description

Converts the duration to an interval or back.

Parameters

`I : Interval;`
Specifies the interval to be converted to a duration.

`D : Duration;`
Specifies the duration to be converted to an interval.

`return Duration;`
Returns the duration from an interval.

`return Interval;`
Returns the interval from a duration.

function Convert_Time

```
function Convert_Time (Date : Calendar.Time) return Time;

function Convert_Time (Date : Time) return Calendar.Time;
```

Description

Converts the time format.

This function converts the time format to or from the time format used in package Calendar.

Parameters

`Date : Calendar.Time;`

Specifies the calendar time to be converted to this time format.

`Date : Time;`

Specifies the time to be converted to the calendar time format.

`return Time;`

Returns the time in this format.

`return Calendar.Time;`

Returns the time in the calendar format.

References

package Calendar

type Date_Format

```
type Date_Format is (Expanded, Month_Day_Year, Day_Month_Year,
                                                   Year_Month_Day, Ada);
```

Description

Defines the set of styles the image of the date can have.

The Image function returns a string that can contain the image of a specified date. A parameter of this type is used to specify the style in which the image of the date is created.

Enumerations

`Ada`

Creates an image of the following style: `83_09_29`

`Day_Month_Year`

Creates an image of the following style: `29-SEP-83`

`Expanded`

Creates an image of the following style: `September 29, 1983`

`Month_Day_Year`

Creates an image of the following style: `09/29/83`

`Year_Month_Day`

Creates an image of the following style: `83/09/29`

constant Day

```
Day : constant Duration := 86_400.0;
```

Description

Defines a constant duration that represents the number of seconds in a day.

type Day_Count

```
type Day_Count is new Integer range 0 .. Integer'Last;
```

Description

Defines a type that represents the number of days.

function Day_Of_Week

```
function Day_Of_Week (T : Calendar.Time) return Weekday;

function Day_Of_Week (T : Time := Time_Utilities.Get_Time) return Weekday;
```

Description

Returns the day of the week corresponding to the indicated time.

Parameters

```
T : Calendar.Time;
T : Time := Time_Utilities.Get_Time;
```

Specifies the time for which to compute the day of week.

```
return Weekday;
```

Returns the day of the week corresponding to the indicated time. Note that Monday has the value 1.

References

type Weekday

type Days

```
type Days is new Calendar.Day_Number;
```

Description

Defines a type that represents the set of days in a month.

function Duration_Until

```
function Duration_Until (T : Time) return Duration;

function Duration_Until (T : Calendar.Time) return Duration;
```

Description

Returns the duration from the current time to the indicated time.

Parameters

```
T : Time;

T : Calendar.Time;
```

Specifies the time from which the duration is to be computed.

```
return Duration;
```

Specifies the duration from the current time to the indicated time.

function Duration_Until_Next

```
function Duration_Until_Next (H : Military_Hours;
                              M : Minutes        := Ø;
                              S : Seconds        := Ø) return Duration;
```

Description

Returns the duration from the current time to the next time with the indicated hour, minute, and second values.

Parameters

```
H : Military_Hours;
```

Specifies the next hour from which the duration is to be computed.

```
M : Minutes := Ø;
```

Specifies the next minute from which the duration is to be computed.

```
S : Seconds := Ø;
```

Specifies the next second from which the duration is to be computed.

```
return Duration;
```

Specifies the duration between the current time and the next time with the indicated hour, minute, and second values.

function Get_Time

```
function Get_Time return Time;
```

Description

Returns the current time.

This function is similar to the Calendar.Clock function, but this function returns the time in a different format.

Parameters

```
return Time;
```

Returns the current time.

References

function Calendar.Clock

constant Hour

```
Hour : constant Duration := 3600.0;
```

Description

Defines a constant duration that represents the number of seconds in an hour.

type Hours

```
type Hours is new Integer range 1 .. 12;
```

Description

Defines a type that represents the number of hours in the A.M. or the P.M.

function Image

```
function Image (Date       : Time;
                Date_Style : Date_Format    := Time_Utilities.Expanded;
                Time_Style : Time_Format    := Time_Utilities.Expanded;
                Contents   : Image_Contents := Time_Utilities.Both
                                                          return String;

function Image (I : Interval) return String;

function Image (D : Duration) return String;

function Image (D : Weekday) return String;
```

Description

Returns the image of the specified time, interval, duration, or weekday.

This function creates a string that represents the specified time. This time can be specified as a value of Time type, Duration type, Interval type, or Weekday type. The first version of the function allows specifying style and content of the string.

Parameters

`Date : Time;`

Specifies the time to create the string.

`Date_Style : Date_Format := Time_Utilities.Expanded;`

Specifies the style of the date portion of the string. This parameter is significant only when the Contents parameter is either Date_Only or Both. The default is the expanded style of the date.

`Time_Style : Time_Format := Time_Utilities.Expanded;`

Specifies the style of the time portion of the string. This parameter is significant only when the Contents parameter is either Time_Only or Both. The default is the expanded style of the time.

`Contents : Image_Contents := Time_Utilities.Both;`

Specifies whether the string should represent the time, the date, or both. The default is both parts.

```
I : Interval;
```

Specifies the interval from which to create the string.

```
D : Duration;
```

Specifies the duration from which to create the string.

```
D : Weekday;
```

Specifies the weekday from which to create the string.

```
return String;
```

Returns the image of the time.

References

type Date_Format

type Image_Contents

type Time_Format

---

type Image_Contents

```
type Image_Contents is (Both, Time_Only, Date_Only);
```

Description

Defines the set of combinations of time and date that can be produced by the Image function.

The Image function returns a string that can contain the image of a date, the image of a time, or both. A parameter of this type specifies which combination is created.

Enumerations

`Both`

Creates an image that combines both time and date. When the time and date format is expanded, the combination looks like this: `11:44:55 PM September 29, 1983`. When the time and date format is Ada, the combination looks like this: `83_09_29_23_44_55`. All combinations of time and date formats are allowed.

`Date_Only`

Creates an image that contains only the date and whose format is governed only by the date format.

`Time_Only`

Creates an image that contains only the time and whose format is governed only by the time format.

type Interval

```
type Interval is
    record
        Elapsed_Days              : Day_Count;
        Elapsed_Hours             : Military_Hours;
        Elapsed_Minutes           : Minutes;
        Elapsed_Seconds           : Seconds;
        Elapsed_Milliseconds      : Milliseconds;
    end record;
```

Description

Defines a segmented version of the Duration type.

function Is_Nil

```
function Is_Nil (Date : Time) return Boolean;

function Is_Nil (Date : Calendar.Time) return Boolean;
```

Description

Checks whether the specified time is nil.

Parameters

`Date : Time;`

Specifies the time to be checked in this time format.

`Date : Calendar.Time;`

Specifies the time to be checked in the calendar format.

`return Boolean;`

Returns the value true when the specified time is nil; otherwise, the function returns false.

type Military_Hours

```
type Military_Hours is new Integer range 0 .. 23;
```

Description

Defines a type that represents the number of hours in a day.

type Milliseconds

```
type Milliseconds is new Integer range 0 .. 999;
```

Description

Defines a type that represents the number of milliseconds in a second.

constant Minute

```
Minute : constant Duration := 60.0;
```

Description

Defines a constant duration that represents the number of seconds in a minute.

type Minutes

```
type Minutes is new Integer range 0 .. 59;
```

Description

Defines a type that represents the number of minutes in an hour.

type Months

```
type Months is (January, February, March, April, May, June, July, August,
                                 September, October, November, December);
```

Description

Defines an enumeration that represents the twelve months of the year.

function Nil

```
function Nil return Time;

function Nil return Calendar.Time;
```

Description

Returns a nil time.

Parameters

```
return Time;
```

Returns the nil time in this time format.

```
return Calendar.Time;
```

Returns the nil time in the calendar format.

type Seconds

```
type Seconds is new Integer range 0 .. 59;
```

Description

Defines a type that represents the number of seconds in a minute.

type Sun_Positions

```
type Sun_Positions is (Am, Pm);
```

Description

Defines an enumeration that represents the two halves of a day.

type Time

```
type Time is
    record
        Year          : Years;
        Month         : Months;
        Day           : Days;
        Hour          : Hours;
        Minute        : Minutes;
        Second        : Seconds;
        Sun_Position  : Sun_Positions;
    end record;
```

Description

Defines a representation for the time.

This representation differs from the one used to represent time in package Calendar. Package Calendar represents time as the year, month, day, and seconds in the day. It represents time in subtypes of integers and duration.

This subtype represents time as the year, month, day, hour, minute, second, and sun position. It uses enumerations for months and sun positions.

References

package Calendar

type Time_Format

```
type Time_Format is (Expanded, Military, Short, Ada);
```

Description

Defines the set of styles the image of the time can have.

The Image function returns a string that can contain the image of a specified time. A parameter of this type is used to specify the style in which the image of the time is created.

Enumerations

`Ada`

Creates an image of the following style: `23_44_55`

`Expanded`

Creates an image of the following style: `11:44:55 PM`

`Military`

Creates an image of the following style: `23:44:55`

`Short`

Creates an image of the following style: `23:44`

function Value

```
function Value (S : String) return Time;

function Value (S : String) return Interval;
```

Description

Converts the string representation of the time into either a time or an interval.

This function takes a string and converts its contents into either a time or an interval. The string has the same allowed formats as the resulting string of the Image function. Input that does not include all fields is assumed to be the current time; that is, 10:30 is assumed to be 10:30:00 A.M. today.

Parameters

```
S : String;
```

Specifies the string to be converted.

```
return Time;
```

Returns the time.

```
return Interval;
```

Returns the interval.

Errors

This function raises the Constraint_Error exception when the input string does not represent a time or interval.

The function raises the Numeric_Error exception if invalid or uninitialized characters are passed to this operation.

type Weekday

```
type Weekday is new Positive range 1 .. 7;
```

Description

Defines a representation for weekdays (Monday is 1).

type Years

```
type Years is new Calendar.Year_Number;
```

Description

Defines a type that represents the set of years.

end Time_Utilities;

RATIONAL

generic function Unchecked_Conversion

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 to be converted to the Target type.

The Target type cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the conversion procedure is called.

A faster, package version of the Unchecked_Conversion function can be found in package Unchecked_Conversions. Note that, although the package version is faster, it will consume more space in the executing program.

generic formal type Source

```
type Source is limited private;
```

Description

Defines the type of object whose bits are to be converted to the Target type.

When the Unchecked_Conversion function is instantiated and then used, the actual types of the source values passed to it and the actual types of the target value returned typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Unchecked_Conversion function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. Typically this error can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

generic formal type Target

```
type Target is limited private;
```

Description

Defines the type of object to which the bits of the Source type are to be converted.

The Target type cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability-_Error exception is raised when the instantiation is elaborated—specifically, when the conversion procedure is called.

When the Unchecked_Conversion function is instantiated and then used, the actual types of the source values passed to it and the actual types of the target value returned typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Unchecked_Conversion function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. Typically this error can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

function Unchecked_Conversion

```
function Unchecked_Conversion (S : Source) return Target;
```

Description

Returns the bit pattern for the source as an object of the Target type.

The type of the target cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the instantiation is elaborated—specifically, when the conversion procedure is called.

The binary representations of the source and target are left-justified. Thus, the leftmost bit of the source object becomes the leftmost bit of the target object. If Target'Size is greater than Source'Size, the target object contains undefined bits in the locations not filled by the source. In most cases, this is undesirable.

If Target'Size is less than Source'Size, the rightmost bits of the source are ignored.

The bits of the source are used from left to right. In some cases, some of the bits will be information on array bounds, discriminants, or the like. For example, if a structured constant is passed as a target to the Unchecked_Conversion function, the constant may contain such information, whereas a variable of the Source type may not. These extra bits thus would give an undesired result or might raise exceptions.

When the Unchecked_Conversion function is instantiated and then used, the actual types of the values passed to it and the actual types of the target value returned typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Unchecked_Conversion function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. Typically this error can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

The following are examples of simple conversions in which the number of bits in the Source and Target types is the same:

- Long_Integer to a record with two Integer fields
- Float to a Long_Integer
- Integer to an array of 32 Booleans
- Record to another record of equal size

Parameters

```
S : Source;
```

Specifies the object to be converted to the Target type. There are no restrictions on the Source type.

```
return Target;
```

Returns an object of the Target type that has the same binary representation as the Source type. This type cannot be an access or task type or contain access or task types as any of its components.

Errors

The System.Capability_Error exception is raised when the function is called if the conversion is impossible because the Target type is not legal (is an access or task type or has components that are access or task types). Note that, for generic subprograms, elaboration of the generic occurs when the instantiation is called.

If the result of an Unchecked_Conversion function does not meet the constraints of the Target type, the Constraint_Error exception is raised. This might occur, for example, if the result of a conversion to a scalar Target type yields a value that exceeds the range constraints for the Target type or if the length of the source and target did not match. See Example 2 below for more information.

The System.Assertion_Error exception may be raised if the object resulting from the conversion has bounds bigger than the Target type allows. See Example 1 below for more information.

The System.Type_Error exception may be raised when the Unchecked_Conversion function is called if the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. Typically, this occurs because the actual types used for the source or the target of the conversion differ from those used in the instantiation of the Unchecked_Conversion function (for example, instantiating using an unconstrained Target type and using Unchecked_Conversion to assign to a variable whose type is a constrained subtype of the Target type). This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed. See Example 2 below for more information.

Example 1

The following program shows the use of the LRM-defined Unchecked_Conversion generic and the Unchecked_Conversions generic package in converting between a discriminated record and a Boolean array. To illustrate the use of both of the generics, the Unchecked_Conversion function is used to convert between the discriminated record and the Boolean array. The package generic Unchecked_Conversions is used to convert back from the Boolean array to the discriminated record.

```
with Unchecked_Conversion;    -- language-defined function from !Lrm
with Unchecked_Conversions;   -- package form from !Tools
with String_Utilities, Io, System;

procedure Conversion is
    -- define discriminated array
    type Vstring (Max_Length : Positive) is
        record
            Length : Positive;
            Contents : String (1 .. Max_Length);
        end record;

    -- define Boolean array
    subtype V20 is Vstring (20);  -- a constrained subtype
    type Bits is array (0 .. 400) of Boolean;

    S : V20;  -- we will be converting between these two objects
    B : Bits;

    -- instantiate both generics
    function Convert1 is new Unchecked_Conversion (V20, Bits);
    package Convert2 is new Unchecked_Conversions.Unchecked_Conversion_Package
                                (Bits, V20);

    generic
        type T is private;
    procedure Display (Object : T);
    -- this procedure produces a binary dump of the object
    procedure Display (Object : T) is
        function Convert_To_Bytes is
           new Unchecked_Conversions.Convert_To_Byte_String  (Source => T);

        procedure Display_Bytes (Value : System.Byte_String) is
            Bytes_Output : Natural := 0;
        begin
            Io.Put_Line ("Object fits in " & Natural'Image (Value'Length) &
                          " bytes.");
            for I in Value'First .. Value'Last loop
                Io.Put (String_Utilities.Number_To_String (Integer (Value (I)),
                                                           Base => 16,
                                                           Width => 2,
                                                           Leading => '0'));
                Bytes_Output := Bytes_Output + 1;
                if Bytes_Output mod 32 = 0 then
                    Io.New_Line;
                elsif Bytes_Output mod 8 = 0 then
                    Io.Put (" ");
                end if;
            end loop;
```

```
        end Display_Bytes;
    begin
        Io.Put_Line ("Object size = " & Integer'Image (Object'Size) & " bits.");
        Display_Bytes (Convert_To_Bytes (Object));
        Io.New_Line;
    end Display;

    procedure Dump is new Display (V20);   -- instantiate display generic for
    procedure Dump is new Display (Bits);  -- each of the two types

begin
    S.Length := 15;  -- initialize S
    S.Contents (1 .. 15) := "000011112222abc";

    -- display initial object
    Io.Put_Line ("Initial VString object:");
    Dump (S);

    -- convert and display result using LRM-defined Unchecked_Conversion
    -- generic
    Io.Put_Line ("Converted bits object:");
    B := Convert1 (S);
    Dump (B);

    -- now modify the bits in it
    Io.Put_Line ("Modified original object:");
    B (256) := True;    -- this changes one of the characters
    S := Convert2.Convert (B);  -- convert bits back into record using
                                -- Unchecked_Conversions generic package
    Dump (S);
    Io.Put_Line (S.Contents (1 .. 15));

end Conversion;
```

The following shows the output generated by the foregoing example:

```
Initial VString object:
Object size =  351 bits.
Object fits in  44 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
64646464C2C4C600 00000000
Converted bits object:
Object size =  401 bits.
Object fits in  51 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
64646464C2C4C600 0000000100000028 000000
Modified original object:
Object size =  351 bits.
Object fits in  44 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
E4646464C2C4C600 00000001
00001111r222abc
```

Example 2

The following example illustrates some of the common errors that can occur when performing unchecked conversions.

```
with Io;
with System;
with Unchecked_Conversion;

procedure Conversion_Errors is


    type S1_Type is range 1 .. 10;  -- 4-bit container
    type T1_Type is range 1 .. 9;   -- 4-bit container

    S1 : S1_Type := 10;
    T1 : T1_Type;

    function Convert is new Unchecked_Conversion (S1_Type, T1_Type);


    type S2_Type is array (1 .. 64) of Boolean;  -- 64-bit container
    type T2_Type is array (Integer range <>) of  -- 128-bit container
                        Long_Integer;            -- because of additional
                                                 -- bounds information

    S2 : S2_Type := (others => True);
    T2 : T2_Type (1 .. 1);

    function Convert is new Unchecked_Conversion (S2_Type, T2_Type);


    type Vstring (Max_Length : Positive) is
        record
            Length : Positive;
            Contents : String (1 .. Max_Length);
        end record;
    subtype V20 is Vstring (20);  -- a constrained subtype

    type Bits is array (0 .. 400) of Boolean;

    S3 : V20;
    T3 : Bits;

    function Convert_To_Bits is new Unchecked_Conversion (V20, Bits);
    function Convert_From_Bits is new Unchecked_Conversion (Bits, Vstring);
    -- note that different actual types were used in the instantiations

begin

    begin
        T1 := Convert (S1);
    exception
        when Constraint_Error =>
            Io.Put_Line ("First conversion raised Contraint_Error");
    end;
    -- this conversion will raise Constraint_Error because the result
    -- of the conversion does not meet the constraints of the Target type


    begin
        T2 := Convert (S2);
    exception
        when System.Assertion_Error =>
```

```
            Io.Put_Line ("Second conversion raised Assertion_Error");
    end;
    -- this conversion will raise Assertion_Error because the object
    -- resulting from the conversion has bounds bigger than the Target
    -- type allows; in this example, it results from putting all 1's
    -- into the bounds information field of an object of an unconstrained
    -- type


    S3.Length := 15;
    S3.Contents (1 .. 15) := (others => ' ');
    T3 := Convert_To_Bits (S3);

    begin
        S3 := Convert_From_Bits (T3);
    exception
        when System.Type_Error =>
            Io.Put_Line ("Third conversion raised Type_Error");
    end;
    -- this conversion will raise the Type_Error exception because the
    -- underlying architectural types used to represent the object S3 and
    -- the Vstring type are not the same

end Conversion_Errors;
```

References

package Unchecked_Conversions

---

end Unchecked_Conversion;

RATIONAL

package Unchecked_Conversions

Package Unchecked_Conversions provides functions for converting objects of one type to objects of another type. It includes generic package Unchecked_Conversion-_Package, which is functionally equivalent to the Unchecked_Conversion function, but it is faster and it provides functions for converting to and from byte strings. Note that, although this package provides faster conversion operations, it will consume more space in the executing program.

The following examples illustrate uses of the operations in this package and highlight some of the common errors that can occur.

Example 1

The following program shows the use of the LRM-defined Unchecked_Conversion generic and the Unchecked_Conversions generic package in converting between a discriminated record and a Boolean array. To illustrate the use of both of the generics, the Unchecked_Conversion function is used to convert between the discriminated record and the Boolean array. The package generic Unchecked_Conversions is used to convert back from the Boolean array to the discriminated record.

```
with Unchecked_Conversion;   -- language-defined function from !Lrm
with Unchecked_Conversions;   -- package form from !Tools
with String_Utilities, Io, System;

procedure Conversion is
    -- define discriminated array
    type Vstring (Max_Length : Positive) is
        record
            Length : Positive;
            Contents : String (1 .. Max_Length);
        end record;

    -- define Boolean array
    subtype V20 is Vstring (20);  -- a constrained subtype
    type Bits is array (0 .. 400) of Boolean;

    S : V20;  -- we will be converting between these two objects
    B : Bits;
```

```
-- instantiate both generics
function Convert1 is new Unchecked_Conversion (V20, Bits);
package Convert2 is new Unchecked_Conversions.Unchecked_Conversion_Package
                           (Bits, V20);

generic
    type T is private;
procedure Display (Object : T);
-- this procedure produces a binary dump of the object
procedure Display (Object : T) is
    function Convert_To_Bytes is
       new Unchecked_Conversions.Convert_To_Byte_String  (Source => T);

    procedure Display_Bytes (Value : System.Byte_String) is
        Bytes_Output : Natural := 0;
    begin
        Io.Put_Line ("Object fits in " & Natural'Image (Value'Length) &
                     " bytes.");
        for I in Value'First .. Value'Last loop
            Io.Put (String_Utilities.Number_To_String (Integer (Value (I)),
                                                         Base => 16,
                                                         Width => 2,
                                                         Leading => '0'));
            Bytes_Output := Bytes_Output + 1;
            if Bytes_Output mod 32 = 0 then
                Io.New_Line;
            elsif Bytes_Output mod 8 = 0 then
                Io.Put (" ");
            end if;
        end loop;
    end Display_Bytes;
begin
    Io.Put_Line ("Object size = " & Integer'Image (Object'Size) & " bits.");
    Display_Bytes (Convert_To_Bytes (Object));
    Io.New_Line;
end Display;
procedure Dump is new Display (V20);   -- instantiate display generic for
procedure Dump is new Display (Bits);  -- each of the two types
```

```
begin
    S.Length := 15;  -- initialize S
    S.Contents (1 .. 15) := "000011112222abc";

    -- display initial object
    Io.Put_Line ("Initial VString object:");
    Dump (S);

    -- convert and display result using LRM-defined Unchecked_Conversion
    -- generic
    Io.Put_Line ("Converted bits object:");
    B := Convert1 (S);
    Dump (B);

    -- now modify the bits in it
    Io.Put_Line ("Modified original object:");
    B (256) := True;     -- this changes one of the characters
    S := Convert2.Convert (B);   -- convert bits back into record using
                                 -- Unchecked_Conversions generic package

    Dump (S);
    Io.Put_Line (S.Contents (1 .. 15));

end Conversion;
```

The following shows the output generated by the foregoing example:

```
Initial VString object:
Object size =  351 bits.
Object fits in  44 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
64646464C2C4C600 00000000
Converted bits object:
Object size =  401 bits.
Object fits in  51 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
64646464C2C4C600 0000000100000028 000000
Modified original object:
Object size =  351 bits.
Object fits in  44 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
E4646464C2C4C600 00000001
00001111r222abc
```

Example 2

The following example illustrates some of the common errors that can occur when performing unchecked conversions.

```
with Io;
with System;
with Unchecked_Conversions;

procedure Conversions_Errors is

    type S1_Type is range 1 .. 10;   -- 4-bit container
    type T1_Type is range 1 .. 9;    -- 4-bit container
    S1 : S1_Type := 10;
    T1 : T1_Type;

    package Convert1 is new Unchecked_Conversions.Unchecked_Conversion_Package
                                (S1_Type, T1_Type);

    type S2_Type is array (1 .. 64) of Boolean;   -- 64-bit container
    type T2_Type is array (Integer range <>) of   -- 128-bit container
                       Long_Integer;              -- because of additional
                                                  -- bounds information

    S2 : S2_Type := (others => True);
    T2 : T2_Type (1 .. 1);

    package Convert2 is new Unchecked_Conversions.Unchecked_Conversion_Package
                                (S2_Type, T2_Type);


    type Vstring (Max_Length : Positive) is
        record
            Length : Positive;
            Contents : String (1 .. Max_Length);
        end record;
    subtype V20 is Vstring (20);  -- a constrained subtype

    S3 : V20;

    function Convert_To_Bytes is
       new Unchecked_Conversions.Convert_To_Byte_String  (V20);
    function Convert_From_Bytes is
       new Unchecked_Conversions.Convert_From_Byte_String  (Vstring);
    -- note that different actual types were used in the instantiations
```

```
begin

    begin
        T1 := Convert1.Convert (S1);
    exception
        when Constraint_Error =>
            Io.Put_Line ("First conversion raised Contraint_Error");
    end;
    -- this conversion will raise Constraint_Error because the result
    -- of the conversion does not meet the constraints of the Target type


    begin
        T2 := Convert2.Convert (S2);
    exception
        when System.Assertion_Error =>
            Io.Put_Line ("Second conversion raised Assertion_Error");
    end;

    -- this conversion will raise Assertion_Error because the object
    -- resulting from the conversion has bounds bigger than the Target
    -- type allows; in this example, it results from putting all 1's
    -- into the bounds information field of an object of an unconstrained
    -- type


    S3.Length := 15;
    S3.Contents (1 .. 15) := (others => ' ');
    declare
        T3 : constant System.Byte_String := Convert_To_Bytes (S3);

    begin
        S3 := Convert_From_Bytes (T3);
    exception
        when System.Type_Error =>
            Io.Put_Line ("Third conversion raised Type_Error");
    end;
    -- this conversion will raise the Type_Error exception because the
    -- underlying architectural types used to represent the object S3 and
    -- the Vstring type are not the same

end Conversions_Errors;
```

RATIONAL

generic package Unchecked_Conversion_Package

Generic package Unchecked_Conversion_Package provides a function for converting objects of one type to objects of another type. This function is functionally equivalent to the Unchecked_Conversion function, but it is faster.

Its formal parameter list is:

```
generic
    type Source is limited private;
    type Target is limited private;
package Unchecked_Conversion_Package  is
    function Convert (S : Source) return Target;
end Unchecked_Conversion_Package;
```

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

The Target type cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the instantiation is elaborated.

function Convert

```
function Convert (S : Source) return Target;
```

Description

Returns the bit pattern for the source as an object of the Target type.

The type of the target cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the instantiation of Unchecked_Conversion_Package is elaborated.

The binary representations of the source and target are left-justified. Thus, the leftmost bit of the source object becomes the leftmost bit of the target object. If Target'Size is greater than Source'Size, the target object contains undefined bits in the locations not filled by the source. In most cases, this is undesirable.

If Target'Size is less than Source'Size, the rightmost bits of the source are ignored.

The bits of the source are used from left to right. In some cases, some of the bits will be information on array bounds, discriminants, or the like. For example, if a structured constant is passed as a target to the Convert function, the constant may contain such information, whereas a variable of the Source type may not. These extra bits thus would give an undesired result or might raise exceptions.

When package Unchecked_Conversion_Package is instantiated and then used, the actual types of the values passed to it and the actual types of the target value returned typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. Typically this error can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

The following are examples of simple conversions in which the number of bits in the Source and Target types is the same:

- Long_Integer to a record with two Integer fields
- Float to a Long_Integer
- Integer to an array of 32 Booleans
- Record to another record of equal size

Parameters

```
S : Source;
```

Specifies the object to be converted to the Target type. There are no restrictions on the Source type.

```
return Target;
```

Returns an object of the Target type that has the same binary representation as the Source type. This type cannot be an access or task type or contain access or task types as any of its components.

Errors

The System.Capability_Error exception is raised on elaboration of the instantiation of the Unchecked_Conversions generic package if the conversion is impossible because the Target type is not legal (is an access or task type or has components that are access or task types).

If the result of a conversion does not meet the constraints of the Target type, the Constraint_Error exception is raised. This might occur, for example, if the result of a conversion to a scalar Target type yields a value that exceeds the range constraints for the Target type or if the length of the source and target did not match. See Example 2 in the package introduction for more information.

The System.Assertion_Error exception may be raised if the object resulting from the conversion has bounds bigger than the Target type allows. See Example 1 in the package introduction for more information.

The System.Type_Error exception may be raised when the Convert function is called if the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. Typically, this occurs because the actual types used for the source or the target of the conversion differ from those used in the instantiation of package Unchecked_Conversion_Package (for example, instantiating using an unconstrained Target type and using Convert to assign to a variable whose type is a constrained subtype of the Target type). This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed. See Example 2 in the package introduction for more information.

Example

Refer to the two examples of the use of this function in the package introduction.

References

function Convert_From_Byte_String

function Convert_To_Byte_String

function Unchecked_Conversion

---

generic formal type Source

```
type Source is limited private;
```

Description

Defines the type of object whose bits are to be converted to the Target type.

When package Unchecked_Conversion_Package is instantiated and then used, the actual types of the source values passed to the Convert function and the actual types of the target value returned typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

generic formal type Target

```
type Target is limited private;
```

Description

Defines the type of object to which the bits of the Source type are to be converted.

The type of the target cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the instantiation of package Unchecked_Conversion_Package is elaborated.

When package Unchecked_Conversion_Package is instantiated and then the Convert function is used, the actual types of the source values passed to it and the actual types of the target value returned typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

end Unchecked_Conversion_Package;

generic function Convert_From_Byte_String

The Convert_From_Byte_String generic function converts byte strings to objects of a given type.

Its formal parameter list is:

```
generic
    type Target is limited private;
function Convert_From_Byte_String (S : System.Byte_String) return Target;
```

The Target type is the type to which the target object bit pattern is to be converted.

function Convert_From_Byte_String

```
function Convert_From_Byte_String (S : System.Byte_String) return Target;
```

Description

Returns the bit pattern for the byte string as an object of the Target type.

The byte string should have been produced by a call to an instantiation of the Convert_To_Byte_String generic function.

The type of the target cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the instantiation is elaborated—specifically, when the conversion function is called.

The binary representations of the byte string and the target are left-justified. Thus, the leftmost bit of the byte string becomes the leftmost bit of the target object. If Target'Size is greater than the length of the byte string, the target object contains undefined bits in the locations not filled by the byte string. In most cases, this is undesirable.

If Target'Size is less than the length of the byte string, the rightmost bits of the byte string are ignored.

The bits of the byte string are used from left to right. In some cases, some of the bits will be information on array bounds, discriminants, or the like. These extra bits must be preserved to avoid undesired results or exceptions.

When the Convert_From_Byte_String function is instantiated and then used, the actual types of the values that are first converted to byte strings using the Convert_To_Byte_String generic function and the actual types of the target value returned typically should be the same as those used in the target instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the original Source and Target types are scalars, but they must be followed if the original Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert_From_Byte_String function is called because the underlying architectural types used to represent the original source and the Target types are not the same, even though the types are compatible in an Ada sense. This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

Parameters

```
S : System.Byte_String;
```

Specifies the byte string from which to create the target. The byte string should have been produced by a call to an instantiation of the Convert_To_Byte_String generic function.

```
return Target;
```

Returns an object of the Target type that has the same binary representation as the byte string. This type cannot be an access or task type or contain access or task types as any of its components.

Errors

The System.Capability_Error exception is raised on elaboration of the generic if the conversion is impossible because the Target type is not legal (is an access or task type or has components that are access or task types). Note that, for generic subprograms, elaboration of the generic occurs when the instantiation is called.

If the result of a conversion does not meet the constraints of the Target type, the Constraint_Error exception is raised. This might occur, for example, if the result of a conversion to a scalar Target type yields a value that exceeds the range constraints for the Target type. See Example 2 in the package introduction for more information.

The System.Assertion_Error exception may be raised if the object resulting from the conversion has bounds bigger than the Target type allows. See Example 1 in the package introduction for more information.

The System.Type_Error exception may be raised when the Convert_From_Byte_String function is called if the underlying architectural types used to represent the original Source and Target types are not the same, even though the types are compatible in an Ada sense. Typically, this occurs because the actual types used for the source or the target of the conversion differ from those used in the instantiation of the Convert_To_Byte_String or the Convert_From_Byte_String functions (for example, instantiating using an unconstrained Target type and using Convert_From_Byte_String to assign to a variable whose type is a constrained subtype of the Target type). This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed. See Example 2 in the package introduction for more information.

Example

Refer to the two examples of the use of this function in the package introduction.

References

function Convert_To_Byte_String

function Unchecked_Conversion

function Unchecked_Conversion_Package.Convert

generic formal type Target

```
type Target is limited private;
```

Description

Defines the type of object to which the bits of the byte string are to be converted.

The type of the target cannot be an access or task type or contain access or task types as any of its components. If these conditions are not met, the System.Capability_Error exception is raised when the instantiation is elaborated—specifically, when the conversion function is called.

When the Convert_To_Byte_String and Convert_From_Byte_String functions are instantiated and then used, the actual types of the source values passed to Convert_To_Byte_String and the actual types of the target value returned from Convert_From_Byte_String typically should be the same as those used in the instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert_From_Byte_String function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

RATIONAL

generic function Convert_To_Byte_String

The Convert_To_Byte_String generic function converts objects of one type to byte strings.

Its formal parameter list is:

```
generic
    type Source is limited private;
function Convert_To_Byte_String (S : Source) return System.Byte_String;
```

The Source type is the type of the source object bit pattern to be converted to a byte string.

function Convert_To_Byte_String

```
function Convert_To_Byte_String (S : Source) return System.Byte_String;
```

Description

Returns the bit pattern for the source as a byte string.

The byte string can then be converted back using the Convert_From_Byte_String function.

The binary representations of the source and the byte string are left-justified. Thus, the leftmost bit of the source object becomes the leftmost bit of the byte string. The byte string length may include additional undefined bits to pad the source object bits to a byte boundary.

The bits of the source are used from left to right. In some cases, some of the bits will be information on array bounds, discriminants, or the like. These extra bits must be preserved to avoid undesired results or exceptions.

When the Convert_To_Byte_String function is instantiated and then used, the actual types of the values passed to it and the actual types of the target value returned from instantiations of Convert_From_Byte_String typically should be the same as those used in the source instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert_From_Byte_String function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

Parameters

```
S : Source;
```

Specifies the object to be converted to a byte string. There are no restrictions on the Source type.

```
return System.Byte_String;
```

Returns the binary representation of the source as a byte string. Thus, the leftmost bit of the source object becomes the leftmost bit of the byte string. The byte string length may include additional undefined bits to pad the source object bits to a byte boundary.

Example

Refer to the two examples of the use of this function in the package introduction.

References

function Convert_From_Byte_String

function Unchecked_Conversion

function Unchecked_Conversion_Package.Convert

---

generic formal type Source

```
type Source is limited private;
```

Description

Defines the type of object whose bits are to be converted to a byte string.

When the Convert_To_Byte_String function is instantiated and then used, the actual types of the source values passed to it and the actual types of the target value returned from instantiations of Convert_From_Byte_String typically should be the same as those used in the source instantiation. Specifically, there should be no differences because of subtypes, derived types, type conversions, constants, aggregates, and so on. These guidelines can be relaxed if the Source and Target types are scalars, but they must be followed if the Source or Target types have discriminants or components that have discriminants.

If there are type mismatch problems, the System.Type_Error exception will be raised when the Convert_From_Byte_String function is called because the underlying architectural types used to represent the Source and the Target types are not the same, even though the types are compatible in an Ada sense. This error usually can be avoided by changing the instantiation to use the actual types required when the conversion is performed.

end Unchecked_Conversions;

generic procedure Unchecked_Deallocation

The Unchecked_Deallocation generic procedure is used to perform unchecked storage deallocation for the designated objects of access types. Unchecked deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components. The Allows_Deallocation generic function can be used to determine whether deallocation can be performed on a particular type.

The formal parameters to the generic procedure are:

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

The Object type is the type of the object for which storage is to be reclaimed. The Name type is an access type that points to objects of the Object type.

For space to be reclaimed, deallocation must be enabled, which can be done by using the Enable_Deallocation library switch or the Enable_Deallocation (X) pragma, where X is the name of the access type for which you want to reclaim storage. Using the library switch enables deallocation for all access types (except those for which deallocation is not supported). Since deallocation adds to the space required for each allocated object, the switch or the pragma should not be used unless you intend to use deallocation. This switch is provided for users who have uploaded code from another type of system and do not want to go back and explicitly add the pragma to each of the access types in the uploaded code. If the library switch is enabled, and you do not want to use deallocation for an access type, you can use the Disable_Deallocation (X) pragma, where X is the type for which deallocation should be disabled. A call to an instantiation of Unchecked_Deallocation will have no effect if the designated type is or contains a task. A call to an instantiation of Unchecked_Deallocation will perform deallocation only on a type that has had deallocation enabled with the pragma or the switch. The pragma will enable deallocation only if it is applied to an access type (not subtype or derived types).

Unchecked Deallocation: R1000

An access base type can be identified as allowing deallocation (with certain restrictions enumerated later). Each element in a collection for such a type contains some overhead to maintain a free list of deallocated elements, currently twice the size of a pointer (usually 24 bits). When allocating in such a collection, the microcode scans the free list for the first deallocated element that is at least as large as the new element (first fit). If no such deallocated element is found, allocation is performed by extending the top of the collection as usual. No coalescing of adjacent deallocated elements occurs; however, any space remaining after the allocation of a new element that exceeds the space required for overhead will be added to the free list.

At the source level, the Enable_Deallocation pragma is used to indicate that a collection will allow deallocation. As described above, the pragma takes the name of an access type as an argument; if the type name is that of a derived type, then the effect of the pragma is the same as that of the pragma with the argument being the parent type. The designated type of the access type cannot contain tasks or segmented heap pointers (or pointers to such types). Any use of unchecked deallocation on an access type to which the pragma has not been applied will have no effect on the corresponding collection.

Example 1

This generic and the pragma are provided for users who want to use deallocation while developing new code. The following example shows how this generic and pragma might be used. T is the access type for which deallocation is enabled.

```
    ...
    ...
type T is access Integer;          -- declare access type T
pragma Enable_Deallocation (T); -- enable deallocation for type T
procedure Free is new Unchecked_Deallocation (Object => Integer, Name => T);
                                   -- instantiate the generic for
                                   -- type T
X : T := new Integer;              -- declare X to be type T and allocate a new
                                   -- object
    ...
    ...
Free (X);                          -- use deallocation to free space
```

If deallocation is performed on either a legal or an illegal object (access to task types), it sets the value of that object to null. If the object is legal, the space will be reclaimed.

Example 2

The Enable_Deallocation pragma can be applied to the generic formal access type, if desired, to suggest that all instantiations should provide an access type for which deallocation is enabled (warnings are produced for instantiations that do not), as shown in the following example:

```
generic
  type T is private;
  type P is access T;
  pragma Enable_Deallocation (P);
package G is
   ...
   ...

end G:

with G;
package X is
   type T1 is new Integer;
   type P1 is access T1;
   pragma Enable_Deallocation (P1); -- instantiate the generic by
   package Y is new G (T1, P1);     -- providing an access type P1
   ...                              -- for which deallocation is enabled
   ...                              -- by the pragma on G's formal P
end X;
```

If X did not contain the Enable_Deallocation pragma or if the library switch was not set in its containing library, unit X would generate errors when it was promoted.

generic formal type Name

```
type Name is access Object;
```

Description

Defines the pointer to the object on which deallocation will be performed.

Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components.

Restrictions

Deallocation is not currently supported for types that contain segmented heap pointers. Many of the types defined by the Environment contain segmented heap pointers and thus cannot be deallocated. Such pointers are not intended for use in application programs and should be used only by Rational technical representatives.

generic formal type Object

```
type Object is limited private;
```

Description

Defines the type of object on which deallocation will performed.

Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components.

Restrictions

Deallocation is not currently supported for types that contain segmented heap pointers. Many of the types defined by the Environment contain segmented heap pointers and thus cannot be deallocated. Such pointers are not intended for use in application programs and should be used only by Rational technical representatives.

procedure Unchecked_Deallocation

```
procedure Unchecked_Deallocation (X : in out Name);
```

Description

Reclaims the storage associated with the object designated by the access value **X**, if possible.

Deallocation is allowed only for types that are not tasks or do not contain tasks or pointers to tasks as any of their components. The Allows_Deallocation generic function can be used to determine whether deallocation can be performed on a particular type.

After the storage for **X** is reclaimed, **X** is set to null. If **X** is null before the call to the Unchecked_Deallocation procedure, the call has no effect. If the Name type did not have deallocation enabled, this call has no effect other than setting **X** to null.

Parameters

```
X : in out Name;
```

Specifies the access value that designates the object for which storage is to be reclaimed. **X** is set to null upon return.

Restrictions

Deallocation is not currently supported for types that contain segmented heap pointers. Many of the types defined by the Environment contain segmented heap pointers and thus cannot be deallocated. Such pointers are not intended for use in application programs and should be used only by Rational technical representatives.

Errors

Note that both the Disable_Deallocation and Enable_Deallocation pragmas should not be used on the same type. If both are used, a warning message appears when the unit is promoted. The same type of warning message is issued if deallocation is enabled on a task type or a type that contains tasks or pointers to tasks. If the unit is semanticized and it contains other semantic errors, the error is underlined along with the other semantic errors. If the unit is semanticized and it contains no other errors, the error is not underlined.

References

generic function Allows_Deallocation

LM, package Switches

Reference Manual for the Ada Programming Language, Appendix F for the R1000 Target, "Pragmas"

---

end Unchecked_Deallocation;

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.

+ function
  Calendar.+ *PT-8*
  Time_Utilities.+ *PT-176*

- function
  Calendar.- *PT-8*
  Time_Utilities.- *PT-177*

< function
  Calendar.< *PT-8*

< generic formal function
  Table_Sort_Generic.< *PT-170*

<= function
  Calendar.<= *PT-8*

= (equal)
  Simple_Status.Equal function PT-135

> function
  Calendar.> *PT-8*

>= function
  Calendar.>= *PT-8*

A

A.M.
  Time_Utilities.Sun_Positions type PT-202

Ada enumeration
  Time_Utilities.Date_Format type PT-180
  Time_Utilities.Time_Format type PT-204

add, *see* Push

add entry, *see* Define

Add procedure
  Queue_Generic.Add *PT-96*
  Set_Generic.Add *PT-112*

address
  System.Null_Address constant PT-166

Address type
  System.Address *PT-164*

Allows_Deallocation function
  Allows_Deallocation.Allows_Deallocation *PT-2*

Allows_Deallocation generic function *PT-1*
  Unchecked_Deallocation generic procedure PT-241
  Unchecked_Deallocation procedure PT-246

array
  Table_Sort_Generic.Element_Array generic formal type PT-172

Ascii package
  Standard.Ascii *PT-161*

Assertion_Error exception
  System.Assertion_Error *PT-164*
    Unchecked_Conversion.Unchecked_Conversion function PT-213
    Unchecked_Conversions.Convert_From_Byte_String function PT-233
    Unchecked_Conversions.Unchecked_Conversion_Pack age.Convert function PT-227

B

beginning, *see* First

Bit constant
  System.Bit *PT-164*

Boolean type
  Standard.Boolean *PT-161*

Both enumeration
  Time_Utilities.Image_Contents type PT-192

byte
  convert from byte string
    Unchecked_Conversions.Convert_From_Byte_String function PT-232
    Unchecked_Conversions.Convert_From_Byte_String generic function . PT-231
  convert to byte string
    Unchecked_Conversions.Convert_To_Byte_String function PT-238
    Unchecked_Conversions.Convert_To_Byte_String generic function . . PT-237

Byte type
  System.Byte *PT-164*

Byte_Size constant
System.Byte_Size . . . *PT-164*

Byte_String type
System.Byte_String . . . *PT-164*

C

Calendar package . . . *PT-5*

Capability_Error exception
System.Capability_Error . . . *PT-164*
Unchecked_Conversion generic function . . . PT-209
Unchecked_Conversion.Target generic formal type . . . PT-211
Unchecked_Conversion.Unchecked_Conversion function . . . PT-212, PT-213
Unchecked_Conversions.Convert_From_Byte_String function . . . PT-232, PT-233
Unchecked_Conversions.Target generic formal type . . . PT-235
Unchecked_Conversions.Unchecked_Conversion_Pack age generic package PT-225
Unchecked_Conversions.Unchecked_Conversion_Pack age.Convert function . . . PT-226, PT-227
Unchecked_Conversions.Unchecked_Conversion_Pack age.Target generic formal type . . . PT-230

car, *see* First

Cardinality function
Concurrent_Map_Generic.Cardinality . . . *PT-10*
Map_Generic.Cardinality . . . *PT-68*

cdr, *see* Rest

Character type
Standard.Character . . . *PT-161*

class
Simple_Status.Condition_Class type . . . PT-130

clear, *see* Make_Empty

Clock function
Calendar.Clock . . . *PT-6*

collating sequence
Table_Sort_Generic.< generic formal function . . . PT-170

collections, *see* Allows_Deallocation, Unchecked_Deallocation

compare, *see* Equal

comparison
Table_Sort_Generic.< generic formal function . . . PT-170

complete, *see* Done

Concurrent_Map_Generic generic package . . . *PT-9*

condition
Simple_Status.Create_Condition procedure PT-132
Simple_Status.Create_Condition_Name function PT-133

condition handling
Simple_Status package PT-127

Condition type
Simple_Status.Condition PT-127, *PT-129*

Condition_Class type
Simple_Status.Condition_Class *PT-130*

Condition_Name type
Simple_Status.Condition_Name *PT-131*

cons, *see* Make

Constraint_Error exception
Concurrent_Map_Generic generic package PT-9
Cardinality function PT-10
Copy procedure PT-11
Define procedure PT-12
Eval function PT-15
Find procedure PT-17
Init procedure PT-19
Is_Empty function PT-22
Make_Empty procedure PT-25
Undefine procedure PT-33
List_Generic generic package PT-49
First function PT-52
Rest function PT-63
Set_First procedure PT-64
Set_Rest procedure PT-65
Map_Generic generic package PT-67
Cardinality function PT-68
Copy procedure PT-69
Define procedure PT-70
Eval function PT-73
Find procedure PT-75
Init procedure PT-77
Is_Empty function PT-80
Make_Empty procedure PT-83
Undefine procedure PT-91
Queue_Generic generic package PT-95
Delete procedure PT-98
First function PT-101
Set_Generic generic package PT-111
Stack_Generic generic package
Next procedure PT-153
Value function PT-159
Standard.Constraint_Error *PT-161*

Constraint_Error exception, continued
Time_Utilities package PT-175
Value function PT-205
Unchecked_Conversion generic function
Unchecked_Conversion function PT-213
Unchecked_Conversions package
Convert_From_Byte_String function PT-233
Unchecked_Conversions.Unchecked_Conversion_Pack age generic package
Convert function PT-227

contents
Time_Utilities.Image_Contents type PT-192

conversion
between different data types
Unchecked_Conversion generic function PT-209
Unchecked_Conversion.Unchecked_Conversion function PT-212
Unchecked_Conversions package PT-219
Unchecked_Conversions.Unchecked_Conversion_Pack age generic package PT-225
Unchecked_Conversions.Unchecked_Conversion_Pack age.Convert function PT-226
from byte string
Unchecked_Conversions.Convert_From_Byte_String function PT-232
Unchecked_Conversions.Convert_From_Byte_String generic function . PT-231
to byte string
Unchecked_Conversions.Convert_To_Byte_String function PT-238
Unchecked_Conversions.Convert_To_Byte_String generic function . . PT-237

conversion, *see also* Image functions for types of particular interest,
Value functions for types of particular interest

Convert function
Time_Utilities.Convert *PT-178*
Unchecked_Conversions.Unchecked_Conversion_Pack age.Convert *PT-226*

Convert_From_Byte_String function
Unchecked_Conversions.Convert_From_Byte_String *PT-232*
Convert_To_Byte_String function PT-238
Source generic formal type PT-240
Target generic formal type PT-235

Convert_From_Byte_String generic function
Unchecked_Conversions.Convert_From_Byte_String *PT-231*

Convert_Time function
Time_Utilities.Convert_Time *PT-179*

Convert_To_Byte_String function
Unchecked_Conversions.Convert_To_Byte_String *PT-238*
Source generic formal type PT-240
Target generic formal type PT-235

Convert_To_Byte_String generic function
Unchecked_Conversions.Convert_To_Byte_String *PT-237*
Convert_From_Byte_String function PT-232

Copy procedure
Concurrent_Map_Generic.Copy *PT-11*
Map_Generic.Copy *PT-69*
Queue_Generic.Copy *PT-97*
Set_Generic.Copy *PT-113*
Stack_Generic.Copy *PT-144*

count
Time_Utilities.Day_Count type PT-182

count, *see also* Cardinality

create entry, *see* Define

Create_Condition procedure
Simple_Status.Create_Condition *PT-132*

Create_Condition_Name function
Simple_Status.Create_Condition_Name *PT-133*

D

data
conversion between different types
Unchecked_Conversion generic function PT-209
Unchecked_Conversion.Unchecked_Conversion function PT-212
Unchecked_Conversions package PT-219
Unchecked_Conversions.Unchecked_Conversion_Pack age generic package PT-225
Unchecked_Conversions.Unchecked_Conversion_Pack age.Convert function PT-226
conversion from byte string
Unchecked_Conversions.Convert_From_Byte_String function PT-232
Unchecked_Conversions.Convert_From_Byte_String generic function . PT-231
conversion to byte string
Unchecked_Conversions.Convert_To_Byte_String function PT-238
Unchecked_Conversions.Convert_To_Byte_String generic function . . PT-237

Date_Format type
Time_Utilities.Date_Format *PT-180*

Date_Only enumeration
Time_Utilities.Image_Contents type PT-192

Day constant
Time_Utilities.Day *PT-181*

Day function
Calendar.Day *PT-6*

Day_Count type
Time_Utilities.Day_Count *PT-182*

Day_Duration subtype
Calendar.Day_Duration *PT-6*

Day_Month_Year enumeration
Time_Utilities.Date_Format type PT-180

Day_Number subtype
  Calendar.Day_Number *PT-6*

Day_Of_Week function
  Time_Utilities.Day_Of_Week *PT-183*

Days type
  Time_Utilities.Days *PT-184*

deallocation
  Allows_Deallocation generic function PT-1
  Allows_Deallocation.Allows_Deallocation function PT-2
  Unchecked_Deallocation generic procedure PT-241
  Unchecked_Deallocation.Unchecked_Deallocation procedure PT-246

Define procedure
  Concurrent_Map_Generic.Define *PT-12*
  Map_Generic.Define *PT-70*

Delete procedure
  Queue_Generic.Delete *PT-98*
  Set_Generic.Delete *PT-114*

delta
  System.Fine_Delta constant PT-165

difference
  Calendar.- function PT-8

digits
  System.Max_Digits constant PT-165

Disable_Deallocation pragma
  Unchecked_Deallocation generic procedure PT-241
  Unchecked_Deallocation.Unchecked_Deallocation procedure PT-247

Display_Message function
  Simple_Status.Display_Message *PT-134*

domain type PT-9, PT-67

Domain_Type generic formal type
  Concurrent_Map_Generic.Domain_Type *PT-13*
  Map_Generic.Domain_Type *PT-71*

Done function
  Concurrent_Map_Generic.Done *PT-14*
    Init procedure PT-19
    Iterator type PT-24
    Next procedure PT-28
  List_Generic.Done *PT-50*
    Init procedure PT-54
    Iterator type PT-57
    Next procedure PT-61

Done function, continued
Map_Generic.Done . . . *PT-72*
Init procedure . . . PT-77
Iterator type . . . PT-82
Next procedure . . . PT-86
Queue_Generic.Done . . . *PT-99*
Init procedure . . . PT-102
Iterator type . . . PT-106
Next procedure . . . PT-108
Set_Generic.Done . . . *PT-115*
Init procedure . . . PT-117
Iterator type . . . PT-121
Next procedure . . . PT-123
Stack_Generic.Done . . . *PT-145*
Init procedure . . . PT-149
Iterator type . . . PT-151
Next procedure . . . PT-153

duplicate, *see* Copy

duration
Calendar.Day_Duration subtype . . . PT-6

Duration type
Standard.Duration . . . *PT-161*
Calendar.Time_Error exception . . . PT-7
Image function . . . PT-190
Interval type . . . PT-193

Duration_Until function
Time_Utilities.Duration_Until . . . *PT-185*

Duration_Until_Next function
Time_Utilities.Duration_Until_Next . . . *PT-186*

E

elapsed duration
Time_Utilities.Duration_Until function . . . PT-185
Time_Utilities.Duration_Until_Next function . . . PT-186

Element generic formal type
List_Generic.Element . . . *PT-51*
Queue_Generic.Element . . . *PT-100*
Set_Generic.Element . . . *PT-116*
Stack_Generic.Element . . . *PT-146*
Table_Sort_Generic.Element . . . *PT-171*

Element_Array generic formal type
Table_Sort_Generic.Element_Array . . . *PT-172*

empty
Concurrent_Map_Generic.Is_Empty function PT-22
Concurrent_Map_Generic.Make_Empty procedure PT-25
List_Generic.Is_Empty function PT-56
Map_Generic.Is_Empty function PT-80
Map_Generic.Make_Empty procedure PT-83
Queue_Generic.Is_Empty function PT-105
Queue_Generic.Make_Empty procedure PT-107
Set_Generic.Is_Empty function PT-119
Set_Generic.Make_Empty procedure PT-122
Stack_Generic.Make_Empty procedure PT-152

empty, *see also* Is_Nil, Nil, Underflow

Empty function
Stack_Generic.Empty *PT-147*

Empty_Stack constant
Stack_Generic.Empty_Stack *PT-148*
Make_Empty procedure PT-152

Enable_Deallocation library switch PT-241

Enable_Deallocation pragma
Unchecked_Deallocation generic procedure PT-241, PT-242
Unchecked_Deallocation.Unchecked_Deallocation procedure PT-247

enumerations
Simple_Status.Condition_Class
Fatal enumeration PT-130
Normal enumeration PT-130
Problem enumeration PT-130
Warning enumeration PT-130
Time_Utilities.Date_Format
Ada enumeration PT-180
Day_Month_Year enumeration PT-180
Expanded enumeration PT-180
Month_Day_Year enumeration PT-180
Year_Month_Day enumeration PT-180
Time_Utilities.Image_Contents
Both enumeration PT-192
Date_Only enumeration PT-192
Time_Only enumeration PT-192
Time_Utilities.Time_Format
Ada enumeration PT-204
Expanded enumeration PT-204
Military enumeration PT-204
Short enumeration PT-204

Equal function
Simple_Status.Equal *PT-135*

error
  Calendar.Time_Error exception PT-7
  Standard.Constraint_Error exception PT-161
  Standard.Numeric_Error exception PT-161
  Standard.Program_Error exception PT-161
  Standard.Storage_Error exception PT-161
  Standard.Tasking_Error exception PT-161
  System.Assertion_Error exception PT-164
  System.Capability_Error exception PT-164
  System.Tape_Error exception PT-167

error, *see also* Constraint_Error exception, Numeric_Error exception, Storage_Error exception

error condition, severity
  Simple_Status.Condition_Class type PT-130

Error function
  Simple_Status.Error *PT-136*

error message
  Simple_Status.Message function PT-140

error message handling
  Simple_Status package PT-127

error name
  Simple_Status.Name function PT-141

error severity
  Simple_Status.Severity function PT-142

Error_Type function
  Simple_Status.Error_Type *PT-138*

Eval function
  Concurrent_Map_Generic.Eval *PT-15*
  Map_Generic.Eval *PT-73*

exceptions
  Calendar package
    Time_Error exception PT-7
  Concurrent_Map_Generic generic package
    Multiply_Defined exception PT-27
    Undefined exception PT-34
  Map_Generic generic package
    Multiply_Defined exception PT-85
    Undefined exception PT-92
  Stack_Generic generic package
    Underflow exception PT-158

exceptions, continued
Standard package
Constraint_Error exception PT-161
Numeric_Error exception PT-161
Program_Error exception PT-161
Storage_Error exception PT-161
Tasking_Error exception PT-161
System package
Assertion_Error exception PT-164
Capability_Error exception PT-164
Tape_Error exception PT-167

exceptions, *see also* Constraint_Error exception, Numeric_Error exception, Storage_Error exception

Expanded enumeration
Time_Utilities.Date_Format type PT-180
Time_Utilities.Time_Format type PT-204

F

Fatal enumeration
Simple_Status.Condition_Class type PT-130

FIFO
Queue_Generic generic package PT-95
Queue_Generic.Queue type PT-109

Find procedure
Concurrent_Map_Generic.Find *PT-16*
Map_Generic.Find *PT-74*

find value, *see* Eval

Fine_Delta constant
System.Fine_Delta *PT-165*

finished, *see* Done

first
List_Generic.Set_First procedure PT-64

First function
List_Generic.First *PT-52*
Queue_Generic.First *PT-101*
Delete procedure PT-98

Float type
Standard.Float *PT-161*

format
Time_Utilities.Date_Format type PT-180
Time_Utilities.Time_Format type PT-204

Free procedure
List_Generic.Free . . . *PT-53*
Is_Empty function . . . PT-56

G

Get_Time function
Time_Utilities.Get_Time . . . *PT-187*

greater than
Calendar.> function . . . PT-8

greater than/equal to
Calendar.>= function . . . PT-8

H

Hash generic formal function
Concurrent_Map_Generic.Hash . . . *PT-18*
Map_Generic.Hash . . . *PT-76*

Hash package . . . *PT-37*

hash table mapping
Concurrent_Map_Generic package . . . PT-9
Map_Generic generic package . . . PT-67

head, *see* First

Hour constant
Time_Utilities.Hour . . . *PT-188*

hours
Time_Utilities.Military_Hours type . . . PT-195

Hours type
Time_Utilities.Hours . . . *PT-189*

I

identical match, *see* Equal

Image function
Time_Utilities.Image . . . *PT-190*
Date_Format type . . . PT-180
Image_Contents type . . . PT-192
Time_Format type . . . PT-204
Value function . . . PT-205

Image_Contents type
Time_Utilities.Image_Contents . . . *PT-192*

increment, *see* Next

index functions
Hash package . . . PT-37

Index generic formal type
  Table_Sort_Generic.Index *PT-173*

Init procedure
  Concurrent_Map_Generic.Init *PT-19*
    Iterator type PT-24
  List_Generic.Init *PT-54*
    Iterator type PT-57
  Map_Generic.Init *PT-77*
    Iterator type PT-82
  Queue_Generic.Init *PT-102*
    Iterator type PT-106
  Set_Generic.Init *PT-117*
    Iterator type PT-121
  Stack_Generic.Init *PT-149*
    Iterator type PT-151

Initialize procedure
  Concurrent_Map_Generic.Initialize *PT-21*
  Map_Generic.Initialize *PT-79*
  Queue_Generic.Initialize *PT-104*
  Set_Generic.Initialize *PT-118*
  Simple_Status.Initialize *PT-139*
    Condition type PT-129

insert, *see* Add, Push

integer
  Hash.Long_Integer_To_Integer function PT-38
  Hash.Pointer_To_Integer function PT-40
  Hash.Pointer_To_Integer generic function PT-39
  Hash.Pointer_To_Long_Integer function PT-44
  Hash.Pointer_To_Long_Integer generic function PT-43
  System.Max_Int constant PT-165
  System.Min_Int constant PT-166

Integer type
  Standard.Integer *PT-161*

Interval type
  Time_Utilities.Interval PT-175, *PT-193*
    Image function PT-190

Is_Empty function
  Concurrent_Map_Generic.Is_Empty *PT-22*
  List_Generic.Is_Empty *PT-56*
  Map_Generic.Is_Empty *PT-80*
  Queue_Generic.Is_Empty *PT-105*
  Set_Generic.Is_Empty *PT-119*

Is_Member function
  Set_Generic.Is_Member *PT-120*

Is_Nil function
Concurrent_Map_Generic.Is_Nil *PT-23*
Map_Generic.Is_Nil *PT-81*
Time_Utilities.Is_Nil *PT-194*

iterate, *see* Init, Next

Iterator type
Concurrent_Map_Generic.Iterator *PT-24*
List_Generic.Iterator *PT-57*
Map_Generic.Iterator *PT-82*
Queue_Generic.Iterator *PT-106*
Set_Generic.Iterator *PT-121*
Stack_Generic.Iterator *PT-151*

L

Length function
List_Generic.Length *PT-58*

less than
Calendar.< function PT-8

less than/equal to
Calendar.<= function PT-8

library switches
Enable_Deallocation PT-241

LIFO
Stack_Generic generic package PT-143
Stack_Generic.Stack type PT-156

linked list
List_Generic generic package PT-49
List_Generic.List type PT-59

List type
List_Generic.List *PT-59*

List_Generic generic package *PT-49*

locate, *see also* Find

Long_Integer type
Standard.Long_Integer *PT-161*

Long_Integer_To_Integer function
Hash.Long_Integer_To_Integer *PT-38*

M

Make function
  List_Generic.Make *PT-60*

Make_Empty procedure
  Concurrent_Map_Generic.Make_Empty *PT-25*
    Is_Empty function PT-22
  Map_Generic.Make_Empty *PT-83*
    Is_Empty function PT-80
  Queue_Generic.Make_Empty *PT-107*
  Set_Generic.Make_Empty *PT-122*
    Is_Empty function PT-119
  Stack_Generic.Make_Empty *PT-152*

management, storage, *see* Allows_Deallocation, Free, Unchecked_Deallocation

mantissa
  System.Max_Mantissa constant PT-165

map generic, concurrent
  Concurrent_Map_Generic generic package PT-9

Map type
  Concurrent_Map_Generic.Map *PT-26*
  Map_Generic.Map *PT-84*

Map_Generic generic package *PT-67*

mapping
  hash table
    Concurrent_Map_Generic package PT-9
    Map_Generic generic package PT-67
  many-to-one
    Hash package PT-37

mapping, *see also* Pair

match, identical, *see* Equal

Max_Digits constant
  System.Max_Digits *PT-165*

Max_Int constant
  System.Max_Int *PT-165*

Max_Mantissa constant
  System.Max_Mantissa *PT-165*

Megabyte constant
  System.Megabyte *PT-165*

membership test
  Set_Generic.Is_Member function PT-120

Memory_Size constant
  System.Memory_Size *PT-165*

message
Simple_Status.Display_Message function PT-134

Message function
Simple_Status.Message *PT-140*

message handling, error
Simple_Status package PT-127

Military enumeration
Time_Utilities.Time_Format type PT-204

Military_Hours type
Time_Utilities.Military_Hours *PT-195*

Milliseconds type
Time_Utilities.Milliseconds *PT-196*

Min_Int constant
System.Min_Int *PT-166*

Minute constant
Time_Utilities.Minute *PT-197*

Minutes type
Time_Utilities.Minutes *PT-198*

Month function
Calendar.Month *PT-6*

Month_Day_Year enumeration
Time_Utilities.Date_Format type PT-180

Month_Number subtype
Calendar.Month_Number *PT-6*

Months type
Time_Utilities.Months *PT-199*

Multiply_Defined exception
Concurrent_Map_Generic.Multiply_Defined PT-9, *PT-27*
Define procedure PT-12
Map_Generic.Multiply_Defined PT-67, *PT-85*
Define procedure PT-70

N

name
Simple_Status.Condition_Name type PT-131
Simple_Status.Create_Condition_Name function PT-133
System.System_Name constant PT-166

Name function
Simple_Status.Name *PT-141*

Name generic formal type
Allows_Deallocation.Name PT-1, *PT-3*
Unchecked_Deallocation.Name PT-241, *PT-244*

Name type
System.Name *PT-166*

Natural subtype
Standard.Natural *PT-161*

next
Time_Utilities.Duration_Until_Next function PT-186

Next procedure
Concurrent_Map_Generic.Next *PT-28*
Iterator type PT-24
List_Generic.Next *PT-61*
Init procedure PT-54
Iterator type PT-57
Map_Generic.Next *PT-86*
Iterator type PT-82
Queue_Generic.Next *PT-108*
Init procedure PT-102
Iterator type PT-106
Set_Generic.Next *PT-123*
Iterator type PT-121
Stack_Generic.Next *PT-153*
Init procedure PT-149
Iterator type PT-151

nil
Concurrent_Map_Generic.Is_Nil function PT-23
Map_Generic.Is_Nil function PT-81
Time_Utilities.Is_Nil function PT-194

nil, *see also* Is_Empty

Nil function
Concurrent_Map_Generic.Nil *PT-29*
Initialize procedure PT-21
List_Generic.Nil *PT-62*
Is_Empty function PT-56
Map_Generic.Nil *PT-87*
Time_Utilities.Nil *PT-200*

Normal enumeration
Simple_Status.Condition_Class type PT-130

null, *see* Is_Empty, Is_Nil, Make_Empty

Null_Address constant
System.Null_Address *PT-166*

number
day
Calendar.Day_Number subtype PT-6
month
Calendar.Month_Number subtype PT-6
year
Calendar.Year_Number subtype PT-7

number, *see also* Cardinality

Numeric_Error exception
Standard.Numeric_Error *PT-161*
Time_Utilities package
Value function PT-205

O

Object generic formal type
Allows_Deallocation.Object PT-1, *PT-4*
Unchecked_Deallocation.Object PT-241, *PT-245*

ordering
Table_Sort_Generic.< generic formal function PT-170

P

P.M.
Time_Utilities.Sun_Positions type PT-202

Pair type
Concurrent_Map_Generic.Pair *PT-30*
Map_Generic.Pair *PT-88*

pointer
Hash.Ptr generic formal type PT-41, PT-45

Pointer_To_Integer function
Hash.Pointer_To_Integer *PT-40*

Pointer_To_Integer generic function
Hash.Pointer_To_Integer *PT-39*

Pointer_To_Long_Integer function
Hash.Pointer_To_Long_Integer *PT-44*

Pointer_To_Long_Integer generic function
Hash.Pointer_To_Long_Integer *PT-43*

Pop procedure
Stack_Generic.Pop *PT-154*

positions
Time_Utilities.Sun_Positions type PT-202

Positive subtype
Standard.Positive *PT-161*

pragmas
  Disable_Deallocation PT-241, PT-247
  Enable_Deallocation PT-241, PT-242, PT-247

printing error messages
  Simple_Status.Display_Message function PT-134

printing status
  Simple_Status.Display_Message function PT-134

Priority subtype
  System.Priority *PT-166*

Problem enumeration
  Simple_Status.Condition_Class type PT-130

Program_Error exception
  Standard.Program_Error *PT-161*

Ptr generic formal type
  Hash.Ptr *PT-41*, *PT-45*

Push procedure
  Stack_Generic.Push *PT-155*

Q

Queue type
  Queue_Generic.Queue *PT-109*

Queue_Generic generic package *PT-95*

R

range type PT-9, PT-67

Range_Type generic formal type
  Concurrent_Map_Generic.Range_Type *PT-31*
  Map_Generic.Range_Type *PT-89*

reclaim, *see* Free

reclaiming storage, *see* Allows_Deallocation, Unchecked_Deallocation

remove, *see* Delete, Pop

remove map entry, *see* Undefine

rest
  List_Generic.Set_Rest procedure PT-65

Rest function
  List_Generic.Rest *PT-63*

rplaca, *see* Set_First

rplacd, *see* Set_Rest

S

search, *see* Find

seconds
  Time_Utilities.Milliseconds type PT-196

Seconds function
  Calendar.Seconds *PT-6*

Seconds type
  Time_Utilities.Seconds *PT-201*

Set type
  Set_Generic.Set *PT-124*
    Initialize procedure PT-118

Set_First procedure
  List_Generic.Set_First *PT-64*

Set_Generic generic package *PT-111*

Set_Rest procedure
  List_Generic.Set_Rest *PT-65*

Severity function
  Simple_Status.Severity *PT-142*

Short enumeration
  Time_Utilities.Time_Format type PT-204

Simple_Status package *PT-127*

size
  System.Byte_Size constant PT-164
  System.Memory_Size constant PT-165
  System.Word_Size constant PT-167

Size generic formal object
  Concurrent_Map_Generic.Size *PT-32*
  Map_Generic.Size *PT-90*

sorting
  Table_Sort_Generic generic procedure PT-169
  Table_Sort_Generic.Table_Sort_Generic procedure PT-174

Source generic formal type
  Unchecked_Conversion.Source *PT-210*
  Unchecked_Conversions.Source *PT-240*
  Unchecked_Conversions.Unchecked_Conversion_Pack age.Source *PT-229*

Split procedure
  Calendar.Split *PT-7*

stack
  Stack_Generic.Empty_Stack constant PT-148

Stack type
Stack_Generic.Stack *PT-156*

Stack_Generic generic package *PT-143*

Standard package *PT-161*

status handling
Simple_Status package PT-127

step, *see* Next

storage management, *see* Allows_Deallocation, Free, Unchecked_Deallocation

storage, reclaiming, *see* Allows_Deallocation, Unchecked_Deallocation

Storage_Error exception
Concurrent_Map_Generic generic package
Initialize procedure PT-21
Map_Generic generic package
Initialize procedure PT-79
Standard.Storage_Error *PT-161*

Storage_Unit constant
System.Storage_Unit *PT-166*

string
System.Byte_String type PT-164

String type
Standard.String *PT-161*

strings
convert from byte
Unchecked_Conversions.Convert_From_Byte_String function PT-232
Unchecked_Conversions.Convert_From_Byte_String generic function . PT-231
convert to byte
Unchecked_Conversions.Convert_To_Byte_String function PT-238
Unchecked_Conversions.Convert_To_Byte_String generic function . . PT-237

sum
Calendar.+ function PT-8

Sun_Positions type
Time_Utilities.Sun_Positions *PT-202*

switches
library
Enable_Deallocation PT-241

System package *PT-163*

System_Name constant
System.System_Name *PT-166*

T

T generic formal type
  Hash.T *PT-42*, *PT-46*

Table_Sort_Generic generic procedure *PT-169*

Table_Sort_Generic procedure
  Table_Sort_Generic.Table_Sort_Generic *PT-174*

tail, *see* Rest

Target generic formal type
  Unchecked_Conversion.Target *PT-211*
  Unchecked_Conversions.Target *PT-235*
  Unchecked_Conversions.Unchecked_Conversion_Pack age.Target *PT-230*

Tasking_Error exception
  Standard.Tasking_Error *PT-161*

throw away, *see* Delete

Tick constant
  System.Tick *PT-167*

time
  Time_Utilities.Convert_Time function PT-179
  Time_Utilities.Get_Time function PT-187

Time type
  Calendar.Time *PT-7*
  Time_Utilities.Time PT-175, *PT-203*
    Image function PT-190

Time_Error exception
  Calendar.Time_Error *PT-7*

Time_Format type
  Time_Utilities.Time_Format *PT-204*

Time_Of function
  Calendar.Time_Of *PT-7*

Time_Only enumeration
  Time_Utilities.Image_Contents type PT-192

Time_Utilities package *PT-175*

Top function
  Stack_Generic.Top *PT-157*

type
domain
Concurrent_Map_Generic.Domain_Type generic formal type PT-13
Map_Generic.Domain_Type generic formal type PT-71
error
Simple_Status.Error_Type function PT-138
range
Concurrent_Map_Generic.Range_Type generic formal type PT-31
Map_Generic.Range_Type generic formal type PT-89

Type_Error exception
System.Type_Error *PT-167*
Unchecked_Conversion.Source generic formal type PT-210
Unchecked_Conversion.Target generic formal type PT-211
Unchecked_Conversion.Unchecked_Conversion function PT-212, PT-213
Unchecked_Conversions.Convert_From_Byte_String function . . PT-232, PT-233
Unchecked_Conversions.Convert_To_Byte_String function PT-238
Unchecked_Conversions.Source generic formal type PT-240
Unchecked_Conversions.Target generic formal type PT-235
Unchecked_Conversions.Unchecked_Conversion_Pack age.Convert function PT-226, PT-227
Unchecked_Conversions.Unchecked_Conversion_Pack age.Source generic formal type PT-229
Unchecked_Conversions.Unchecked_Conversion_Pack age.Target generic formal type PT-230

U

Unchecked_Conversion function
Unchecked_Conversion.Unchecked_Conversion *PT-212*, PT-219, PT-225

Unchecked_Conversion generic function *PT-209*

Unchecked_Conversion_Pack age generic package
Unchecked_Conversions.Unchecked_Conversion_Pack age PT-219, *PT-225*

Unchecked_Conversions package *PT-219*

Unchecked_Deallocation generic procedure *PT-241*
Allows_Deallocation generic function PT-1
Allows_Deallocation.Allows_Deallocation function PT-2

Unchecked_Deallocation procedure
Unchecked_Deallocation.Unchecked_Deallocation *PT-246*

Undefine procedure
Concurrent_Map_Generic.Undefine *PT-33*
Map_Generic.Undefine *PT-91*

Undefined exception
Concurrent_Map_Generic.Undefined *PT-34*
Eval function PT-15
Undefine procedure PT-33
Map_Generic.Undefined *PT-92*
Eval function PT-73
Undefine procedure PT-91

Underflow exception
Stack_Generic.Underflow PT-143, *PT-158*
Pop procedure PT-154
Top function PT-157

uninitialized, *see* Is_Nil, Nil

unit
System.Storage_Unit constant PT-166

until
Time_Utilities.Duration_Until function PT-185
Time_Utilities.Duration_Until_Next function PT-186

utilities
Time_Utilities package PT-175

V

value, *see* Eval

Value function
Concurrent_Map_Generic.Value *PT-35*
Init procedure PT-19
Iterator type PT-24
List_Generic.Value *PT-66*
Init procedure PT-54
Iterator type PT-57
Map_Generic.Value *PT-93*
Init procedure PT-77
Iterator type PT-82
Queue_Generic.Value *PT-110*
Init procedure PT-102
Iterator type PT-106
Set_Generic.Value *PT-125*
Init procedure PT-117
Iterator type PT-121
Stack_Generic.Value *PT-159*
Init procedure PT-149
Iterator type PT-151
Time_Utilities.Value *PT-205*

W

Warning enumeration
  Simple_Status.Condition_Class type PT-130

week
  Time_Utilities.Day_Of_Week function PT-183

Weekday type
  Time_Utilities.Weekday *PT-206*
    Image function PT-190

Word_Size constant
  System.Word_Size *PT-167*

Y

Year function
  Calendar.Year *PT-7*

Year_Month_Day enumeration
  Time_Utilities.Date_Format type PT-180

Year_Number subtype
  Calendar.Year_Number *PT-7*

Years type
  Time_Utilities.Years *PT-207*

RATIONAL

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

____
____
____
____
____
____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

____
____
____
____

Indicate any additions or changes you would like to see in the index.

____
____
____
____

How much experience have you had with the Rational Environment?

6 months or less _______ 1 year _______ 3 years or more ______

How much experience have you had with the Ada programming language?

6 months or less _______ 1 year _______ 3 years or more ______

Name (optional)____________________________ Date______________
Company ____
Address ____
City ____________________ State ____________ ZIP Code ____________

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043

Rational Environment Reference Manual, Programming Tools (PT), 8001A-29