# REPORT DOCUMENTATION PAGE

Form Approved

OPM No.



| 1. AGENCY USE (Leave | 2. REPORT | 3. REPORT TYPE AND DATES |
|---|---|---|
| | | |

**4. TITLE AND**: Rational Software Corporation, 940608W1.11357, Compiler: Apex 1.4.1

**5. FUNDING**



**6.** Authors: Wright-Patterson, AFB, Dayton, OH, USA

**7. PERFORMING ORGANIZATION NAME (S) AND**: Ada Validating Facility
Language Control Facility ASD/SCEL, Bldg. 676, Rm. 135
Wright-Patterson AFB, Dayton OH 45433

**8. PERFORMING ORGANIZATION**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND** :
Ada Joint Program Office 701 S. Courthouse Rd.
DISA Arlington, VA
Code TXEA 22204-2199

**10. SPONSORING/MONITORING AGENCY**

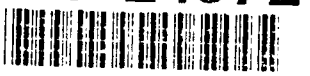


**11. SUPPLEMENTARY**

**12a. DISTRIBUTION/AVAILABILITY**: Approved for Public Release; distribution unlimited

**12b. DRISTRIBUTION**





**13.** ***(Maximum 200***
Host and Target: SPARCstation 10/51 (under SunOS 2.3)

**14. SUBJECT**: Ada Programming Language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability Validation Testing, Ada Validation Office, Ada Validation Facility

**15. NUMBER OF**

**16. PRICE**

| 17 SECURITY CLASSIFICATION | 18. SECURITY | 19. SECURITY CLASSIFICATION | 20. LIMITATION OF |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED |

NSN

AVF Control Number: AVF-VSR-585.0694
Date VSR Completed: 5 July 1994
94-03-07-RAT



Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 940608W1.11357
Rational Software Corporation
Apex, 1.4.1
Sun SPARCstation 10/51 under Solaris 2.3

(Final)


Prepared By:
Ada Validation Facility
645 CCSG/SCSL
Wright-Patterson AFB OH 45433-5707

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 8 June 1994.

Compiler Name and Version: Apex, 1.4.1

Host Computer System: Sun SPARCstation 10/51 under Solaris 2.3

Target Computer System: Same as host

Customer Agreement Number: 94-03-07-RAT

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 940608W1.11357 is awarded to Rational Software Corporation. This certificate expires two years after MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.

Ada Validation Facility
Dale E. Lange
Technical Director
645 CCSG/SCSL
Wright-Patterson AFB OH 45433-5707

for Ada Validation Organization
Director, Computer and Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Donald J. Reifer
Director, AJPO
Defense Information Systems Agency,
Center for Information Management

DECLARATION OF CONFORMANCE

Customer: Rational Software Corporation

Ada Validation Facility: Computer Operations Division
Information Systems and Technology Center
Wright-Patterson AFB, OH 45433-6503

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: Apex 1.4.1

Host and Target Computer System: SparcStation 10/51 SunOS 5.3 (Solaris 2.3)

Customer's Declaration

I, the undersigned, representing Rational Software Corporation, declare that Rational Software Corporation has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration.

Steve Zeigler

Date: 5/16/94

Steve Zeigler
2800 San Tomas Expressway
Santa Clara, CA 95051-0951

TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro92]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

## 1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

## 1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro92] Ada Compiler Validation Procedures, Version 3.1, Ada Joint Program Office, August 1992.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

## 1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

## 1.4 DEFINITION OF TERMS

| | |
|---|---|
| Ada Compiler | The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof. |
| Ada Compiler Validation Capability (ACVC) | The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report. |
| Ada Implementation | An Ada compiler with its host computer system and its target computer system. |
| Ada Joint Program Office (AJPO) | The part of the certification body which provides policy and guidance for the Ada certification system. |
| Ada Validation Facility (AVF) | The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation. |
| Ada Validation Organization (AVO) | The part of the certification body that provides technical guidance for operations of the Ada certification system. |
| Compliance of an Ada Implementation | The ability of the implementation to pass an ACVC version. |
| Computer System | A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units. |

| | |
|---|---|
| Conformity | Fulfillment by a product, process, or service of all requirements specified. |
| Customer | An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed. |
| Declaration of Conformance | A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized. |
| Host Computer System | A computer system where Ada source programs are transformed into executable form. |
| Inapplicable test | A test that contains one or more test objectives found to be irrelevant for the given Ada implementation. |
| ISO | International Organization for Standardization. |
| LRM | The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>." |
| Operating System | Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible. |
| Target Computer System | A computer system where the executable form of Ada programs are executed. |
| Validated Ada Compiler | The compiler of a validated Ada implementation. |
| Validated Ada Implementation | An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro92]. |
| Validation | The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation. |
| Withdrawn test | A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language. |

# CHAPTER 2

# IMPLEMENTATION DEPENDENCIES

## 2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of witndrawn tests is 22 November 1993.

| | | | | | |
|---|---|---|---|---|---|
| B27005A | E28005C | B28006C | C32203A | C34006D | C35507K |
| C35507L | C35507N | C35507O | C35507P | C35508I | C35508J |
| C35508M | C35508N | C35702A | C35702B | C37310A | B41308B |
| C43004A | C45114A | C45346A | C45612A | C45612B | C45612C |
| C45651A | C46022A | B49008A | B49008B | A54B02A | C55B06A |
| A74006A | C74308A | B83022B | B83022H | B83025B | B83025D |
| C83026A | B83026B | C83041A | B85001L | C86001F | C94021A |
| C97116A | C98003B | BA2011A | CB7001A | CB7001B | CB7004A |
| CC1223A | BC1226A | CC1226B | BC3009B | BD1B02B | BD1B06A |
| AD1B08A | BD2A02A | CD2A21E | CD2A23E | CD2A32A | CD2A41A |
| CD2A41E | CD2A87A | CD2B15C | BD3006A | BD4008A | CD4022A |
| CD4022D | CD4024B | CD4024C | CD4024D | CD4031A | CD4051D |
| CD5111A | CD7004C | ED7005D | CD7005E | AD7006A | CD7006E |
| AD7201A | AD7201E | CD7204B | AD7206A | BD8002A | BD8004C |
| CD9005A | CD9005B | CDA201E | CE2107I | CE2117A | CE2117B |
| CE2119B | CE2205B | CE2405A | CE3111C | CE3116A | CE3118A |
| CE3411B | CE3412B | CE3607B | CE3607C | CE3607D | CE3812A |
| CE3814A | CE3902B | | | | |

## 2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

| | |
|---|---|
| C24113L..Y (14 tests) | C35705L..Y (14 tests) |
| C35706L..Y (14 tests) | C35707L..Y (14 tests) |
| C35708L..Y (14 tests) | C35802L..Z (15 tests) |
| C45241L..Y (14 tests) | C45321L..Y (14 tests) |
| C45421L..Y (14 tests) | C45521L..Z (15 tests) |
| C45524L..Z (15 tests) | C45621L..Z (15 tests) |
| C45641L..Y (14 tests) | C46012L..Z (15 tests) |

The following 21 tests check for the predefined type SHORT_INTEGER; for this implementation, there is no such type:

| | | | | |
|---|---|---|---|---|
| C35404B | B36105C | C45231B | C45304B | C45411B |
| C45412B | C45502B | C45503B | C45504B | C45504E |
| C45611B | C45613B | C45614B | C45631B | C45632B |
| B52004E | C55B07B | B55B09D | B86001V | C86006D |
| CD7101E | | | | |

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

| | | | | |
|---|---|---|---|---|
| C35404C | C45231C | C45304C | C45411C | C45412C |
| C45502C | C45503C | C45504C | C45504F | C45611C |
| C45613C | C45614C | C45631C | C45632C | B52004D |
| C55B07A | B55B09C | B86001W | C86006C | CD7101F |

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

LA3004A..B, EA3004C..D, and CA3004E..F (6 tests) check pragma INLINE for procedures and functions; this implementation does not support pragma INLINE.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

| Test | File Operation | Mode | File Access Method |
|---|---|---|---|
| CE2102D | CREATE | IN_FILE | SEQUENTIAL_IO |
| CE2102E | CREATE | OUT_FILE | SEQUENTIAL_IO |
| CE2102F | CREATE | INOUT_FILE | DIRECT_IO |
| CE2102I | CREATE | IN_FILE | DIRECT_IO |
| CE2102J | CREATE | OUT_FILE | DIRECT_IO |
| CE2102N | OPEN | IN_FILE | SEQUENTIAL_IO |
| CE2102O | RESET | IN_FILE | SEQUENTIAL_IO |
| CE2102P | OPEN | OUT_FILE | SEQUENTIAL_IO |
| CE2102Q | RESET | OUT_FILE | SEQUENTIAL_IO |
| CE2102R | OPEN | INOUT_FILE | DIRECT_IO |
| CE2102S | RESET | INOUT_FILE | DIRECT_IO |
| CE2102T | OPEN | IN_FILE | DIRECT_IO |
| CE2102U | RESET | IN_FILE | DIRECT_IO |
| CE2102V | OPEN | OUT_FILE | DIRECT_IO |
| CE2102W | RESET | OUT_FILE | DIRECT_IO |
| CE3102E | CREATE | IN_FILE | TEXT_IO |
| CE3102F | RESET | Any Mode | TEXT_IO |
| CE3102G | DELETE | ———— | TEXT_IO |

| | | | |
|---|---|---|---|
| CE3102I | CREATE | OUT_FILE | TEXT_IO |
| CE3102J | OPEN | IN_FILE | TEXT_IO |
| CE3102K | OPEN | OUT_FILE | TEXT_IO. |

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A check operations on text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

## 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 122 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

| | | | | | |
|---|---|---|---|---|---|
| B22003A | B22003B | B22004A | B22004B | B22004C | B22005K |
| B22005L | B23002A | B23004A | B23004B | B24001A | B24001B |
| B24001C | B24005A | B24005B | B24007A | B24009A | B24104A |
| B24204A | B24204B | B24204C | B24204D | B24204E | B24204F |
| B24205A | B24206A | B24206B | B25002A | B25002B | B26001A |
| B26002A | B26005A | B28003A | B28003C | B29001A | B2A003A |
| B2A003B | B2A003C | B2A003D | B2A003E | B2A003F | B2A004A |
| B2A005A | B2A005B | B2A007A | B2A021A | B32103A | B33101A |
| B33201B | B33202B | B33203B | B33301A | B33301B | B35101A |
| B36002A | B37106A | B37205A | B37307B | B38003A | B38003B |
| B38009A | B38009B | B38103A | B38103B | B38103C | B38103D |
| B38103E | B41201A | B44001A | B44004A | B44004B | B44004C |
| B45205A | B48002A | B48002D | B51001A | B53003A | B55A01A |
| B61005A | B64006A | B67001A | B67001B | B67001C | B67001D |
| B67001H | B71001A | B71001G | B71001M | B74104A | B74307B |
| B83E01C | B83E01D | B85008G | B85008H | B91001H | B95001D |
| B95003A | B95004A | B95063A | BA1101E | BB1006B | BB3005A |
| BC1013A | BC1109A | BC1109B | BC1109C | BC1109D | BC1201A |
| BC1206A | BC1303F | BC2001D | BC2001E | BC3003B | BC3005B |

BC3013A BD2B14A BD2C14A BE2210A BE2413A

BC3204D and BC3205C were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CE3804H was graded passed by Evaluation Modification as directed by the AVO. This test requires that the string "-3.525" can be read from a file using FLOAT_IO and that an equality comparison with the numeric literal '-3.525' will evaluate to TRUE; however, because -3.525 is not a model number, this comparison may evaluate to FALSE (LRM 4.9:12). This implementation's compile-time and run-time evaluation algorithms differ; thus, this check for equality fails and Report.Failed is called at line 81, which outputs the message "WIDTH CHARACTERS NOT READ." All other checks were passed.

# CHAPTER 3

# PROCESSING INFORMATION

## 3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

Jerry Rudisin
Rational Software Corporation
2800 San Tomas Expressway
Santa Clara CA 95051-0951
(408) 496-3712

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

## 3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro92].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

| | | |
|---|---|---|
| a) Total Number of Applicable Tests | 3750 | |
| b) Total Number of Withdrawn Tests | 104 | |
| c) Processed Inapplicable Tests | 115 | |
| d) Non-Processed I/O Tests | 0 | |
| e) Non-Processed Floating-Point Precision Tests | 201 | |
| f) Total Number of Inapplicable Tests | 316 | (c+d+e) |
| g) Total Number of Tests for ACVC 1.11 | 4170 | (a+b+f) |

## 3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The validation tape was loaded on a machine acting as a file server. The files were accessed via automounted NFS.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled, linked and executed on the host computer system. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

| Option \| Switch | Effect |
|---|---|
| -listing_directory <dir> | To produce a listing in the specified directory <dir>. |
| -compile | To syntactically and semantically analyze a source file and produce an Ada unit (or units) if correct. |
| -goal linked | Produce the object code and linked executable for an Ada main program. |

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

## APPENDIX A

## MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN—also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

| Macro Parameter | Macro Value |
|---|---|
| $MAX_IN_LEN | 254 — Value of V |
| $BIG_ID1 | (1..V-1 => 'A', V => '1') |
| $BIG_ID2 | (1..V-1 => 'A', V => '2') |
| $BIG_ID3 | (1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A') |
| $BIG_ID4 | (1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A') |
| $BIG_INT_LIT | (1..V-3 => '0') & "298" |
| $BIG_REAL_LIT | (1..V-5 => '0') & "690.0" |
| $BIG_STRING1 | '"' & (1..V/2 => 'A') & '"' |
| $BIG_STRING2 | '"' & (1..V-1-V/2 => 'A') & '1' & '"' |
| $BLANKS | (1..V-20 => ' ') |
| $MAX_LEN_INT_BASED_LITERAL | "2:" & (1..V-5 => '0') & "11:" |
| $MAX_LEN_REAL_BASED_LITERAL | "16:" & (1..V-7 => '0') & "F.E:" |

```
$MAX_STRING_LITERAL    '"' & (1..V-2 => 'A') & '"'
```

The following table lists all of the other macro parameters and their respective values.

| Macro Parameter | Macro Value |
|---|---|
| $ACC_SIZE | 32 |
| $ALIGNMENT | 1 |
| $COUNT_LAST | 1000000000 |
| $DEFAULT_MEM_SIZE | 2147483647 |
| $DEFAULT_STOR_UNIT | 8 |
| $DEFAULT_SYS_NAME | SPARC_SOLARIS |
| $DELTA_DOC | 0.000_000_000_465_661_287_307_739_257_812_5 |
| $ENTRY_ADDRESS | SYSTEM.TO_ADDRESS (30) |
| $ENTRY_ADDRESS1 | SYSTEM.TO_ADDRESS (31) |
| $ENTRY_ADDRESS2 | SYSTEM.TO_ADDRESS (2) |
| $FIELD_LAST | 2147483647 |
| $FILE_TERMINATOR | ' ' |
| $FIXED_NAME | NO_SUCH_TYPE |
| $FLOAT_NAME | NO_SUCH_TYPE |
| $FORM_STRING | "" |
| $FORM_STRING2 | "CANNOT_RESTRICT_FILE_CAPACITY" |
| $GREATER_THAN_DURATION | 1.0 |
| $GREATER_THAN_DURATION_BASE_LAST | 131_073.0 |
| $GREATER_THAN_FLOAT_BASE_LAST | 1.0E308 |
| $GREATER_THAN_FLOAT_SAFE_LARGE | 3.3E38 |

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E308

$HIGH_PRIORITY          255

$ILLEGAL_EXTERNAL_FILE_NAME1
                        BAD/_CHARACTERS

$ILLEGAL_EXTERNAL_FILE_NAME2
                        CONTAINS/_WILDCARDS

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST");

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006D1.TST");

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -1.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE
```

MACRO PARAMETERS

| | |
|---|---|
| $NAME_LIST | SPARC_SOLARIS |
| $NAME_SPECIFICATION1 | X2120A |
| $NAME_SPECIFICATION2 | X2120B |
| $NAME_SPECIFICATION3 | X3119A |
| $NEG_BASED_INT | 16#FFFFFFFE# |
| $NEW_MEM_SIZE | 2147483647 |
| $NEW_STOR_UNIT | 8 |
| $NEW_SYS_NAME | SPARC_SOLARIS |
| $PAGE_TERMINATOR | ASCII.FF |
| $RECORD_DEFINITION | NEW INTEGER; |
| $RECORD_NAME | NO_SUCH_MACHINE_CODE_TYPE |
| $TASK_SIZE | 32 |
| $TASK_STORAGE_SIZE | 8192 |
| $TICK | (1.0/60.0) |
| $VARIABLE_ADDRESS | FCNDECL.ADDRESS0 |
| $VARIABLE_ADDRESS1 | FCNDECL.ADDRESS1 |
| $VARIABLE_ADDRESS2 | FCNDECL.ADDRESS2 |
| $YOUR_PRAGMA | EXPORT_OBJECT |

# APPENDIX B

## COMPILATION AND LINKER OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

# Rational Apex Compiler and Linker Switches

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| APEX_CLOSURE | Comp | String | Yes | | Used to determine additional units to be analyzed. The empty string (" ") is equivalent to specifying **Installed**. If the **APEX_CONFIGURATION** switch is specified, the closure set is computed from units in the views of the configuration, otherwise the set is computed from units in the imported views | " ", **Installed**, **Coded**, **Linked** |
| APEX_COMPILE_ CONFIGURATION | Comp | Boolean | Yes | False | Compile all views in the configuration. | |
| APEX_FIRST_ERROR | Comp | Boolean | Yes | False | Continue past the first unit with errors. If True, command will stop after the first unit containing an error. | **True**, **False** |
| APEX_INTO | Comp | String | Yes | | The directory into which the files will be parsed. Must name a view or a directory in a view. If blank, the current directory is used. | " ", view or directory in view |
| APEX_NEW_ RELEASE | Comp | Boolean | Yes | False | Recompiles the Ada units in each view, converting to new DIANA, CG attribute, and program library formats | **True**, **False** |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| CODE_LISTING | Comp | Boolean | Yes | False | Create a **.asm** file, which includes assembly code and source annotations, during compilation. If **True**, the file is created in the **.Rational/Compilation** directory for the view that contains the Ada main program. | **True, False** |
| COMPILE | Comp | Boolean | Yes | False | Causes the compiler to parse and compile source files. | **True, False** |
| COMPILER_KEY | Comp | Pathname | No | Installation dependent | The compiler to use, which provides for platform-specific semantic checking and code generation. Set by installation. Do not change. | Compiler pathname |
| CONFIGURATION | Comp | Pathname | Yes | | Configuration to use during a link. If blank, the imports are used. Can be used to specify units to compile beyond those found in the code closure in the imported views. | Configuration pathname |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| ELABORATION_ ORDER_LISTING | Comp | Boolean | Yes | False | Create a file containing a listing of the elaboration order of the units in its closure is created when a main program is linked. Elaboration-order listings can be created only for main programs. Elaboration-order files are stored in the **.Rational/Compilation** subdirectory of the view and have the name **foo.2.elab_order** for a main unit named **foo.2.ada**. | **True, False** |
| FLAG_INEVITABLE_ EXCEPTIONS | Comp | Boolean | No | False | Control the handling of any statically determinable situation that is certain to raise an exception when executed, *such as an out-of-bounds assignment.* This switch is overridden by the **REJECT_INEVITABLE_EXCEPTIONS** switch. When **REJECT_INEVITABLE_EXCEPTIONS** is **True**, this switch has no effect. If **REJECT_INEVITABLE_EXCEPTIONS** is **False** and this switch is:<br>■ **True**, such situations are reported with warning messages<br>■ **False**, such situations are ignored. | **True, False** |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| IGNORE_INVALID_ REP_SPECS | Comp | Boolean | No | False | Control the handling of invalid or unsupported representation specifications. Representation specifications are considered invalid if they do not conform to the restrictions specified in Chapter 10, "LRM Appendix F: Implementation-Dependent Characteristics." When **True**, this switch overrides the **IGNORE_UNSUPPORTED_REP_ SPECS** switch. In this case, both invalid and unsupported representation specifications are reported with warning messages in the output window and are otherwise ignored. When **False**, the treatment of invalid representation specifications depends on the setting of the **IGNORE_UNSUPPORTED_REP_ SPECS** switch. | **True, False** |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| IGNORE_ UNSUPPORTED_ REP_SPECS | Comp | Boolean | No | False | Control the handling of unsupported representation specifications. This switch is overridden by the **IGNORE_INVALID_REP_SPECS** switch. When **IGNORE_INVALID_REP_SPECS** is **True**, this switch has no effect. If **IGNORE_INVALID_REP_SPECS** is **False** and this switch is:<br>■ **True**, unsupported representation specifications are reported with warning messages in the output window and are otherwise ignored<br>■ **False**, unsupported representation specifications are treated as errors, causing analysis of the units that contain them to fail.<br>**Note:** For most purposes, the **IGNORE_INVALID_REP_ SPECS** switch and the **IGNORE_UNSUPPORTED_ REP_SPECS** switch should have the same value. | **True, False** |

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| INCREMENTAL_LINK | Comp | Boolean | Yes | False | Attempt to use incremental features of the platform linker.<br><br>**Note:** This switch applies only to AIX. | **True, False** |
| LISTING_DIRECTORY | Comp | Pathname | Yes | "" | To produce a listing in the specified directory | Listing directory pathname |
| NON_ADA_LINKAGE | Comp | String | Yes | | The arguments to pass to the target linker. This can be used to specify object files and archive libraries for non-Ada program units that will be included when an Ada main program is linked. | |
| OPTIMIZATION_ LEVEL | Comp | Integer | Yes | 0 | The optimization level to use for compiling. 0 is fastest compilation, 2 is best code. | **0 - 2** |
| OPTIMIZATION_ OBJECTIVE | Comp | String | Yes | Time | The optimization objective, either Time or Space, to be used for any compilation unit that does not contain a pragma **Optimize**. | **Time, Space** |
| PROFILING | Comp | String | Yes | | The type of profiling to use when preparing the code. | **" ", None, Prof, Gprof** |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| REJECT_BAD_LRM_ PRAGMAS | Comp | Boolean | No | False | Control the handling of illegal Ada pragmas. When **True**, illegal Ada pragmas are treated as errors, thus causing analysis of the units that contain them to fail. When **False**, illegal Ada pragmas are reported with warning messages in the output window and are otherwise ignored. | **True, False** |
| REJECT_BAD_ RATIONAL_PRAGMAS | Comp | Boolean | No | False | Control the handling of illegal Rational-defined pragmas. When **True**, illegal Rational pragmas are treated as errors, thus causing analysis of the units that contain them to fail. When **False**, illegal Rational pragmas are reported with warning messages in the output window and are otherwise ignored. | **True, False** |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| REJECT_ INEVITABLE_ EXCEPTIONS | Comp | Boolean | No | False | Control the handling of any statically determinable situation that is certain to raise an exception when executed, such as an out-of-bounds assignment. When **True**, this switch overrides the **FLAG_INEVITABLE_EXCEPTIONS** switch and inevitable exceptions are treated as errors, thus causing analysis of the units that contain them to fail.When **False**, the treatment of inevitable exceptions depends on the setting of the **FLAG_INEVITABLE_ EXCEPTIONS** switch. | **True, False** |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# Rational Apex Compiler and Linker Switches (cont.)

| Switch Name | Area | Command Type | Option | Default Value | Description | Range |
|---|---|---|---|---|---|---|
| REJECT_ STATEMENT_ PROMPTS | Comp | Boolean | No | False | The compiler will allow you to code Ada units that contain **[statement]** prompts. | **True, False** |
| REJECT_ UNDEFINED_ PRAGMAS | Comp | Boolean | No | False | Control the handling of any pragmas not defined in the LRM or in the PRM. When **True**, undefined pragmas are treated as errors, thus causing analysis of the units that contain them to fail. When **False**, undefined pragmas are reported with warning messages in the output window and are otherwise ignored. | **True, False** |
| RUNTIMES | Comp | Pathname | Yes | | *The pathname for the Rational Ada* runtimes to use at link time if the default version is not appropriate. In most cases, the default switch value (" ") is the correct one to use.<br><br>**Note:** This switch is for use only at the direction of Rational technical support. | *Runtime* pathname |

**Legend:** Comp - Compilation; OM - Object Management; PP - Pretty Printing

Click on underlined text for more information.

# APPENDIX C

## APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
   ..........

    type Integer is range -2147483648 .. 2147483647;

    type Float is digits 6
          range -3.40282E+38 .. 3.40282E+38;

    type Long_Float is digits 15
          range -1.79769313486231E+308 .. 1.79769313486231E+308;

    type Duration is delta  0.000061035156
          range -131072.00000000 .. 131071.999938964843750000000;

   ..........
end STANDARD;
```

*Chapter 10*

# *LRM Appendix F: Implementation-Dependent Characteristics*

This chapter provides information as required by the LRM Appendix F. The implementation-dependent characteristics of Rational Ada are described in the following sections:



## Pragmas

This section provides:

- General notes about pragma error handling
- A table of implementation dependencies in the predefined pragmas from LRM Annex B
- A table of implementation-defined pragmas
- Detailed descriptions of each of the implementation-defined pragmas starting on page 83

### Error Handling for Pragmas

A pragma whose existence, placement, or arguments do not correspond to those allowed is ignored, by default, by the

compiler and the runtime system. This means that a warning is generated if the compiler detects such an error, but in any event the compilation completes successfully.

Several Apex context switches allow you, however, to specify whether to treat certain classes of invalid pragmas as errors that prevent successful compilation rather than as warnings. See Appendix A, "Switches," for details on the following switches:

- Reject_Bad_Lrm_Pragmas: Affects the handling of illegal LRM-defined pragmas
- Reject_Bad_Rational_Pragmas: Affects the handling of illegal Rational implementation-defined pragmas
- Reject_Undefined_Pragmas: Affects the handling of pragmas defined neither in the LRM nor in this Appendix F

If more than one of the same pragma is specified where it is not appropriate to do so (for example, two pragma Mains on the same unit), the first one is used and the others generate warnings at compile time.

### References

- pragma warnings, LRM 2.8(9), 2.8(11)

## Predefined Pragmas

For each pragma defined in Annex B of the LRM, Table 10-1 describes the extent to which Rational Ada supports it.

***Table 10-1 Predefined Pragmas from LRM Annex B***

| Predefined Pragma | Level of Support |
|---|---|
| Controlled | Always implicitly in effect because the implementation does not support automatic garbage collection |
| Elaborate | As described in Annex B |
| Inline | Has no effect |
| Interface | As described in Annex B; must be used in conjunction with pragmas Import_Procedure and Import_Function |

*Table 10-1 Predefined Pragmas from LRM Annex B (Continued)*

| Predefined Pragma | Level of Support |
|---|---|
| List | As described in Annex B |
| Memory_Size | Has no effect |
| Optimize | Has an effect only when located in the outermost scope, where it applies to the entire compilation unit: if not used, the value SPACE is assumed. See "Setting the Optimization Objective" on page 21 |
| Pack | As described in Annex B; see "Concepts for Object Sizes" on page 67 |
| Page | As described in Annex B |
| Priority | As described in Annex B and LRM 9.8(2); the default is 127 |
| Shared | As given in Annex B; has an effect only for integer, enumeration, access, and fixed types |
| Storage_Unit | Has no effect |
| Suppress | As described in Annex B |
| System_Name | Has no effect (there is only one enumeration literal in the type System.System_Name) |

## Implementation-Defined Pragmas

Table 10-2 summarizes all implementation-defined pragmas in Rational Ada. Each pragma is described in more detail in the following subsections.

*Table 10-2 Implementation-Defined Pragmas*

| Implementation-Defined Pragma | Description |
|---|---|
| Assert | Raises an exception if a specified Boolean expression evaluates to False at run time |
| Collection_Policy | Controls memory allocation for the collection designated by an access type |
| Export_Function | Creates a global symbol for an Ada function so that it can be called by non-Ada code |
| Export_Object | Creates a global symbol for an Ada object so that it can be referenced by non-Ada code |
| Export_Procedure | Creates a global symbol for an Ada procedure so that it can be called by non-Ada code |
| Generic_Policy | Tells the compiler how to generate code for a generic and its instantiations |
| Import_Function | Associates the global symbol for a non-Ada function with an Ada name so that an Ada subprogram can call the function |
| Import_Object | Associates the global symbol for a non-Ada object with an Ada name, so that an Ada subprogram can reference the object |
| Import_Procedure | Associates the global symbol for a non-Ada procedure with an Ada name, so that an Ada subprogram can call the procedure |
| Initialize | Specifies that default initialization be carried out for an imported object or an object referenced by an address clause |
| Instance_Policy | Specifies replicated code for specific instantiations of a shared-code generic |
| Main | Designates an Ada main unit and specifies aspects of its run-time behavior |
| Must_Be_Constrained | Indicates whether formal private and limited private types within a generic formal part must be constrained |

*Table 10-2 Implementation-Defined Pragmas (Continued)*

| Implementation-Defined Pragma | Description |
|---|---|
| Signal_Handler | Installs a procedure as a UNIX signal handler |
| Suppress_All | Suppresses all permitted runtime checks |
| Suppress_Elaboration-_Checks | Suppresses elaboration checks for a specific compilation unit |

## Pragma Assert

Raises an exception if a specified Boolean expression evaluates to False at run time. The syntax is:

```
pragma ASSERT
   ([PREDICATE  =>]  boolean_expression);
```

### Arguments

- **Predicate**: The Boolean expression to be evaluated at run time.

### Usage

When the pragma is encountered at run time, the Boolean expression is evaluated. If the result is False, the System.Assertion_Error exception is raised; if the result is True, no action is taken.

This pragma can appear anywhere that a declaration or statement is allowed.

## Pragma Collection_Policy

Controls memory allocation for the collection designated by an access type. The syntax is:

```
pragma COLLECTION_POLICY
   (ACCESS_TYPE       =>   access_type,
    INITIAL_SIZE      =>   integer_expression
    [, EXTENSIBLE     =>   boolean_expression]
    [, EXTENSION_SIZE =>   integer_expression]);
```

## Arguments

- **Access_Type**: The access type on which to perform storage management.
- **Initial_Size**: The size in storage units of the initial collection that is created for an access type. A negative value is treated as 0.
- **Extensible**: Specifies whether the collection can be extended. If True, sets the extension size to the default or specified value of Extension_Size; otherwise, Extension_Size is ignored. The default value is True.
- **Extension_Size**: The minimum number of storage units by which the collection will be extended, when needed, if it is extensible. A negative value is treated as 0. The default value is 4,096 bytes.

## Usage

The pragma must appear in the same declarative region as the access type to which it applies, after the access type's declaration and before any forcing occurrence of the access type. If the access type is a private type, the pragma must appear in the private part after the complete access-type declaration. If the pragma appears outside the specified areas, it is ignored.

A description and example of the pragma's application are provided in "Managing Storage for Access Types" on page 63.

## Notes

- The arguments must be specified using named association.
- Only one Collection_Policy pragma is allowed per access type. If more than one is specified, the first is applied and the rest are ignored.
- When an access type has an associated 'Storage_Size clause, any Collection_Policy pragma for that access type is ignored. This occurs because a statement of the form:

  ```
  for X'STORAGE_SIZE use size;
  ```

  is functionally equivalent to:

```
pragma COLLECTION_POLICY
   (ACCESS_TYPE     =>   X,
    INITIAL_SIZE    =>   size,
    EXTENSIBLE      =>   FALSE);
```

- If Initial_Size is nonpositive and Extensible is False, attempting to execute an allocator of the access type raises the Storage_Error exception.

### References

- access type, LRM 3.8
- allocator, LRM 4.8
- collection, LRM 3.8
- forcing occurrence, LRM 13.1(6)
- Storage_Error, LRM 11.1
- Storage_Size, LRM 13.7.2

## Pragmas Export_Function, Export_Object, and Export_Procedure

Creates a global symbol for an Ada subprogram (function or procedure) or object so that it can be called or referenced by non-Ada code. The syntax is:

```
pragma EXPORT_FUNCTION
   ([INTERNAL             =>]  internal_name
    [, [EXTERNAL          =>]  "external_name"]
    [, [PARAMETER_TYPES   =>]  parameter_type_list]
    [, [RESULT_TYPE       =>]  type_mark]
    [, [LANGUAGE          =>]  language_name]);

pragma EXPORT_OBJECT
   ([INTERNAL    =>]  internal_name
    [, [EXTERNAL=>]  "external_name"]);

pragma EXPORT_PROCEDURE
   ([INTERNAL             =>]  internal_name
    [, [EXTERNAL          =>]  "external_name"]
    [, [PARAMETER_TYPES   =>]  parameter_type_list]
    [, [LANGUAGE          =>]  language_name]);
```

## Arguments

- **Internal**: The Ada simple name of the Ada subprogram or object to be exported. For functions, this can also be an operator symbol.
- **External**: An optional string literal that specifies the global symbol name to be created by the Ada compiler. This name must obey the naming conventions for the host operating system's object-module format, and therefore it may differ from the internal subprogram or object name. If an external name is not specified, the internal name is used for the symbol name.
- **Parameter_Types**: A parenthesized, comma-separated list of type and/or subtype names that describes the parameter-type profile of an exported subprogram. If the subprogram has no parameters, the list can consist of the single word Null. This optional argument may be required when the Internal argument specifies an overloaded subprogram. See "Usage," below.
- **Result_Type**: The result-type profile of an exported function. This optional argument may be required when the Internal argument specifies an overloaded function. See "Usage," below.
- **Language**: The name of the language in which the calling code is written. The only language name currently supported is C; always use this value when exporting to C or C++. Other language names are ignored, so this argument can be omitted when exporting to a language other than C or C++.

## Usage

Use these export pragmas to create global symbolic names for Ada subprograms that will be called—or Ada objects that will be referenced—from non-Ada code. The linker will use these symbols to resolve intermodule references.

If the internal subprogram name is overloaded, you must supply enough information for the compiler to determine unambiguously which subprogram to export. Specify the Parameter_Types (and/or, for functions, the Result_Type) so

that the compiler can construct the parameter- and/or result-type profile of the subprogram.

***Caution:*** *Exporting a* subprogram *does not export the mechanism used by the compiler to perform elaboration checks. A call from another language to an exported subprogram with an unelaborated body may produce unpredictable results when the subprogram references an object that is itself unelaborated.*

***Caution:*** *Accesses to Ada* objects *by non-Ada code are inherently unsafe; the compiler and runtime system cannot guarantee the integrity of such exported objects. It is the developer's responsibility to ensure that the code that accesses an exported object properly interprets and maintains the underlying structure of the object.*

## Notes

- An export The pragma can appear only at the place of a declarative item in a declarative part or package specification; the subprogram or object to which it applies must have been declared by an earlier declarative item of the same declarative part or package specification.
- An exported subprogram must:
  - *Not* be a generic
  - Be declared in a static scope; that is, it must not be inside any subprogram, task, generic unit, or block statement
- An exported object must:
  - *Not* be in a generic unit
  - Be a variable
  - Be declared in a static scope; that is, it must not be inside any subprogram, task, generic unit, or block statement
  - Have a static size; that is, its subtype must be one of:
    - A scalar type or subtype
    - An array subtype with static index constraints whose component size is static
    - An undiscriminated record type or subtype

### References for Subprograms

- elaboration of a library unit, LRM 10.5
- order of elaboration, LRM 3.9
- overloading, LRM 8.3
- parameter and result type profile, LRM 6.6
- "Calling an Ada Subprogram from C" on page 8

### References for Objects

- limited private type, LRM 7.4.4
- private type, LRM 7.4
- "Sharing Global Objects" on page 14

## Pragma Generic_Policy

Tells the compiler how to generate code for a generic package or subprogram and its instantiations. The syntax is:

```
pragma GENERIC_POLICY
   ([GENERIC_UNIT=>] simple_name,
    [CODE        =>] REPLICATED | SHARED);
```

### Arguments

- **Generic_Unit**: The simple name of the generic package or subprogram to which the pragma applies.
- **Code**: A keyword that specifies whether all instantiations should share the code in one common routine (Shared), or whether each instantiation should be coded separately (Replicated).

### Usage

See "Setting Shared or Replicated Generic Policy" on page 22.

### Notes

- Use pragma Instance_Policy to override the shared Generic-_Policy for one or more instantiations of a generic package or subprogram. See "Pragma Instance_Policy" on page 93.
- The compiler treats all generics as Replicated unless otherwise specified with pragma Generic_Policy.

- The pragma can appear only at the place of a declarative item in a declarative part or package specification; the generic to which it applies must have been declared by an earlier declarative item of the same declarative part or package specification.
- Any generics appearing in Apex interface views must be shared, since the compiler cannot access the generic body to use as a template for coding replicated instantiations.

### References

- generic instantiation, LRM 12.3
- generic package, LRM 12.1
- generic subprogram, LRM 12.1
- simple name, LRM 4.1

## Pragmas Import_Function, Import_Object, and Import_Procedure

Associates an Ada name with the global symbol for a non-Ada subprogram (function or procedure) or object so that an Ada subprogram can call the subprogram or reference the object. The syntax is:

```
pragma IMPORT_FUNCTION
   ([INTERNAL             =>]  internal_name
    [, [EXTERNAL          =>]  "external_name"]
    [, [PARAMETER_TYPES   =>]  parameter_type_list]
    [, [RESULT_TYPE       =>]  type_mark]
    [, [MECHANISM         =>]  mechanism_list]);

pragma IMPORT_OBJECT
   ([INTERNAL    =>]  internal_name
    [, [EXTERNAL=>]  "external_name"]);

pragma IMPORT_PROCEDURE
   ([INTERNAL             =>]  internal_name
    [, [EXTERNAL          =>]  "external_name"]
    [, [PARAMETER_TYPES   =>]  parameter_type_list]
    [, [MECHANISM         =>]  mechanism_list]);
```

### Arguments

- **Internal**: The Ada simple name of the non-Ada subprogram or object to be imported.

- **External**: An optional string literal that specifies the global symbol name to be created by the Ada compiler. Since other languages may enforce non-Ada-compatible naming conventions, the external symbol may differ from the internal subprogram or object name. If an external name is not specified, the internal name is used for the symbol name.
- **Parameter_Types**: A parenthesized, comma-separated list of type and/or subtype names that describes the parameter-type profile of an imported subprogram. If the subprogram has no parameters, the list can consist of the single word Null. This optional argument may be required when the Internal argument specifies an overloaded subprogram. See "Usage," below.
- **Result_Type**: The result-type profile of an imported function. This optional argument may be required when the Internal argument specifies an overloaded function. See "Notes," below.
- **Mechanism**: A parenthesized, comma-separated list of parameter-passing mechanisms for the parameters passed by a subprogram. If the imported subprogram has parameters, then the Mechanism argument is required; otherwise, do not include Mechanism. There must be a one-to-one correspondence between the passed parameters and the mechanisms. The supported mechanisms are:
  - **Value**: The corresponding parameter is passed by value.

    ***Note:*** *When interfacing with C or C++, only scalars can be passed by value; Mechanism must always be Value and the corresponding Ada parameter must be an* in *parameter for scalars.*

  - **Reference**: The corresponding parameter is passed by reference; that is, its address is passed. This applies to records and arrays in C and C++ and to C++ constant reference parameters.

  If all of the imported subprogram's parameters are passed with the same mechanism, you can specify a single occurrence of the mechanism without parentheses.

## Usage

Use the import pragmas to supply more information about a non-Ada subprogram specified with pragma Interface or a non-Ada object to be referenced by Ada code.

Every imported subprogram must be described both by pragma Interface and by an import pragma, in that order. Pragma Interface is ignored if there is no corresponding import pragma, or if the import pragma contains errors.

If the internal Ada subprogram name is overloaded, you must supply enough information for the compiler to determine unambiguously which subprogram is being imported. Specify the Parameter_Types (and/or, for functions, the Result_Type) so that the compiler can construct the parameter- and/or result-type profile of the subprogram.

***Caution:*** *Accesses to non-Ada objects from Ada code are inherently unsafe; the compiler and runtime system cannot guarantee the integrity of such imported objects. It is the developer's responsibility to ensure that the code that accesses an imported object properly interprets and maintains the underlying structure of the object.*

## Notes

- An import The pragma can appear only at the place of a declarative item in a declarative part or package specification; the subprogram or object to which it applies must have been declared by an earlier declarative item of the same declarative part or package specification.
- An import pragma must not refer to a generic subprogram.
- An imported object must:
  - *Not* be in a generic unit
  - Be a variable declared at the outermost level of a library-package specification or body
  - Have a static size; that is, its subtype must be one of:
    - A scalar type or subtype
    - An array subtype with static index constraints whose component size is static
    - An undiscriminated record type or subtype

- To assign an imported object a default initial value, use pragma Initialize. See "Pragma Initialize" on page 92.

### References for Subprograms

- interface to other languages, LRM 13.9
- pragma Interface, LRM 13.9
- scalar types, LRM 3.3
- "Calling a Non-Ada Subprogram from Ada" on page 6

### References for Objects

- limited private type, LRM 7.4.4
- private type, LRM 7.4
- "Sharing Global Objects" on page 14

## Pragma Initialize

Specifies that default initialization be carried out for an imported variable or a variable referenced by an address clause. The syntax is:

```
pragma INITIALIZE
   (simple_name);
```

### Arguments

- ***simple_name***: The variable to which default initialization is to be applied.

### Usage

When a program imports a variable object or declares a variable with an address clause, the compiler assumes that this variable previously existed. The compiler makes no attempt to assign a default (initial) value to this variable, because the variable might already contain a valid value or might be given an initial value by some other program. By default, the compiler does not perform any initialization on:

- Variables designated by address clauses
- Imported variable objects; see "Pragmas Import_Function, Import_Object, and Import_Procedure" on page 89

Pragma Initialize tells the compiler to assign an appropriate default value to the variable—for example, setting pointers and pointer fields to null, record fields to the initial values present in the record type definition, and discriminants to their proper values. Hence, the variable must not have an explicit initial value.

No additional storage space is allocated because valid variables already exist.

The referenced variable must:

- Have been declared earlier in the same declarative part
- Be an array or record
- Have an associated address clause, or it must have been imported using pragma Import_Object before the occurrence of pragma Initialize
- *Not* have an explicit initial value

### Example

Pragma Initialize can be used to request that pointers be set to Null or that record fields be given some starting value.

### References

- address clause, LRM 13.5
- default initialization, LRM 3.2.1

## Pragma Instance_Policy

Specifies how to generate code for specific instantiations of a generic. The syntax is:

```
pragma INSTANCE_POLICY
   ([INSTANTIATION =>]   simple_name,
    [CODE          =>]   REPLICATED | SHARED);
```

### Arguments

- **Instantiation**: The simple name of the specific instantiation to which the pragma applies.
- **Code**: A keyword whose value can be Replicated or Shared.

## Usage

Use pragma Instance_Policy to specify whether to generate replicated or shared code for specific instantiations of generics.

The following example illustrates the use of the pragma:

```
-- EXCHANGE_I and EXCHANGE_R use the common shared
-- code. EXCHANGE_S uses its own replicated code.
--
generic
   type Sometype is private;
procedure Swap(X, Y: in out Sometype);
pragma Generic_Policy(Swap, Shared);
--
procedure Exchange_R is new Swap(Sometype => Real);
procedure Exchange_I is new Swap(Sometype => Integer);
subtype S is String(1..100);
procedure Exchange_S is new Swap(Sometype => S);
pragma Instance_Policy(Exchange_S, Replicated);
```

## Notes

- The pragma is ignored if the instantiation refers to a generic in an Apex interface view.
- The pragma and the named instantiation must occur within the same declarative part or package specification.
- The instantiation must occur before the pragma.
- If the instantiation argument refers to several preceding overloaded subprogram instantiations, the pragma applies to all of them.
- Only one pragma Instance_Policy can be applied to each instantiation.

## References

- "Pragma Generic_Policy" on page 88
- "Setting Shared or Replicated Generic Policy" on page 22
- generic instantiation, LRM 12.3
- generic package, LRM 12.1
- generic subprogram, LRM 12.1
- simple name, LRM 4.1

## Pragma Main

Designates an Ada main unit and determines some aspects of its runtime behavior. The syntax is:

```
pragma MAIN
[ ([DETECT_DEADLOCK          => boolean_expression,]
   [HEAP_SIZE                => static_integer_expression,]
   [NONBLOCKING_IO           => boolean_expression,]
   [POSIX_COMPLIANT          => boolean_expression,]
   [PREEMPTIVE_SCHEDULING    => boolean_expression,]
   [SIGNAL_STACK_SIZE        => static_integer_expression,]
   [STACK_SIZE               => static_integer_expression,]
   [TASK_PRIORITY_DEFAULT    => priority_expression,]
   [TASK_STACK_SIZE_DEFAULT
                             => static_integer_expression,]
   [TIME_SLICE               => duration_expression]) ];
```

### Arguments

- **Detect_Deadlock**: Specifies whether the Rational Ada runtime system should diagnose deadlock situations in the program. If True, the runtime system will print a diagnosis of what is causing the tasks to block when deadlock occurs. If False, the program simply hangs. The default is False.
- **Heap_Size**: A nonnegative static integer expression that specifies how much space to allocate for the heap, in bytes, when the main unit begins execution. If this argument is specified, no additional space is allocated to the heap after initialization; requests for more heap space raise Storage_Error.

  If not specified, heap space is allocated dynamically as needed until space is exhausted and Storage_Error is raised.
- **Nonblocking_Io**: Specifies whether I/O should block all tasks in the program. If True, only the task performing the I/O blocks; if False, the entire program blocks. For a description of limitations and operation, see "I/O in Tasking Programs" on page 34 and "Using Blocking and Nonblocking I/O" on page 58. The default is False.

- `Posix_Compliant`: Specifies whether certain behavior described by the IEEE Portable Operating System Interface (POSIX) is required. If True, the following operational characteristics of programs compiled and linked under Rational Apex are affected:
  - The program can control only those UNIX signals explicitly allowed by POSIX.5 3.3.3.1 (those not "reserved for the Ada implementation").
  - The program cannot install an interrupt-entry task to handle UNIX signals that the runtime system uses, nor can it install both an interrupt-entry task and an Ada procedural signal handler for the same signal (POSIX.5 3.3.2.1(963)).
  - The default values for the Form-parameter fields in the Ada-predefined I/O packages are the POSIX.5 values rather than the Apex values, as described in "Field Defaults" on page 50.

  The default is True.
- `Preemptive_Scheduling`: Specifies whether preemptive (asynchronous) task scheduling takes place. If True, all tasks spawned by the main program are scheduled preemptively. The default is False. Task scheduling is described in Chapter 5, "Ada Tasking in UNIX."
- `Signal_Stack_Size`: An integer expression greater than or equal to 2,048 (2 Kb) that specifies the size of the signal stack, in bytes. The Rational Ada runtime system uses this stack for handling runtime signals, and the debugger uses this stack for special type display. If not specified, the default signal stack size is 64 Kb. When the program is run under the debugger, the default stack size is increased to 2 Mb.
- `Stack_Size`: A static integer expression greater than or equal to 2,048 (2 Kb) that specifies the size of the main task stack, in bytes. If not specified, the default stack size is 2 Mb.
- `Task_Priority_Default`: An expression of type System.Priority that specifies the priority for any task without a pragma Priority. The default is the same as the main task's priority; if the main task is not given a priority, the default is 127.

- **Task_Stack_Size_Default**: A static integer expression greater than or equal to 2,048 (2 Kb) that specifies the size, in bytes, of the stack for any task without a 'Storage_Size representation clause. The default is 64 Kb.
- **Time_Slice**: A nonnegative expression of type Standard-.Duration that determines the quantity of time to allocate to an executing task. By default, or if the value is zero, no time slicing is used. This has an effect only in conjunction with preemptive scheduling; otherwise, it is ignored.

## Usage

Use pragma Main after the end of the unit body of any parameterless library-unit procedure to designate it as a main program.

Pragma Main can have two effects. It:

- Causes the unit to be linked automatically if it is in the directory or view for which you have requested linking; main units without pragma Main are not linked unless explicitly requested.
- Permanently specifies the size of various code sections and the mode of operation for the executable program resulting from a link.

## Example

Use pragma Main as shown:

```
procedure Show_Main is
begin
   Do_Something;
end Show_Main;
pragma Main (Stack_Size => 10*1024); --Change to 10 Kb
```

## Notes

- All arguments can be specified using Apex session switches to change the options dynamically at runtime. The switches have the same names as the arguments, in all uppercase, with the prefix **APEX_**. Explicit use of an argument on pragma Main overrides the switch values.

### References

- library unit, LRM 10.1
- main program, LRM 10.1
- heap and stack allocation, "Miscellaneous Memory Management" on page 64

## Pragma Must_Be_Constrained

Indicates whether formal private and limited private types within a generic formal part must be constrained or have default values. The syntax is:

```
pragma MUST_BE_CONSTRAINED
   (condition_list);
```

### Arguments

- ***condition_list***: A comma-separated list of conditions that specifies a set of types and whether each set must be constrained or have default values. Each element of the *condition* list has the format:

  ```
  [condition =>] type_id_list
  ```

  where:

  - ***condition***: Can be either YES or NO. If omitted, the default is YES. Determines the setting for all types in the following type ID list.
  - ***type_id_list***: A comma-separated list of formal private or limited private types. These types must be defined in the same formal part as the pragma.

### Usage

Use pragma Must_Be_Constrained to specify how you intend to use the formal parameters in a generic specification.

Each condition controls the types in the following type ID list, until the next occurrence of a condition. Consider this example:

```
pragma Must_Be_Constrained
   (Type_1, NO=>Type_2, Type_3, YES=>Type_4, Type_5);
```

At the beginning of the list, a condition is not specified, so YES is assumed; hence, Type_1 is constrained. NO controls the

following type ID list, which includes Type_2 and Type_3; hence, they are unconstrained. YES controls the remaining type ID list, so Type_4 and Type_5 are constrained.

### Notes

If the condition NO is specified, any use in the body that requires a constrained type will generate a semantic error. If YES is specified, any instantiations that contain actual parameters that require constrained types will generate semantic errors if the actual parameters are not constrained and have no default values.

### References

- constrained private type, LRM 7.4.2
- generic formal type, LRM 12
- matching rules for formal private types, LRM 12.3.2
- limited private type, LRM 7.4.4
- private type as generic formal type, LRM 12.1.2

## Pragma Signal_Handler

Installs an Ada procedure as a UNIX signal handler. The syntax is:

```
pragma SIGNAL_HANDLER
   (NAME      => simple_name,
    SIGNAL    => integer_expression);
```

### Arguments

- **Name**: The simple Ada name of the signal-handling procedure.
- **Signal**: A nonstatic integer expression specifying the UNIX signal number to be handled by the specified procedure.

### Usage

Elaboration of the pragma has the effect of installing the specified procedure as a signal handler for the given signal; subsequent occurrences of the specified signal will cause the specified procedure to be invoked.

The pragma and the procedure body must occur in the same declarative part, with the pragma following the procedure body. This prevents the installation of a procedure whose body has not yet been elaborated.

See "Ada Procedural Signal Handlers" on page 43 for details on the construction of the procedure.

### References

- simple name, LRM 4.1
- declarative part, LRM 3.9

## Pragma Suppress_All

Suppresses all permitted runtime checks. The syntax is:

```
pragma SUPPRESS_ALL;
```

### Arguments

None.

### Usage

Use pragma Suppress_All to create the same effect as all of the following:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Division_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Storage_Check);
pragma Suppress (Range_Check);
```

### Notes

- Pragma Suppress_All has no effect in a package specification.
- The pragma must appear immediately within a declarative part.

### References

- suppressing checks, LRM 11.

## Pragma Suppress_Elaboration_Checks

Suppresses all elaboration checks in a given compilation unit. The syntax is:

```
pragma Suppress_Elaboration_Checks;
```

### Arguments

None.

### Usage

Use pragma Suppress_Elaboration_Checks after the end of the unit body of any compilation unit to suppress elaboration checks for all subprograms in that unit. This is equivalent to placing a named pragma Suppress (Elaboration_Check) on each subprogram in the unit.

### References

- suppressing checks, LRM 11.7

# Attributes

Table 10-3 summarizes all implementation-defined attributes in Rational Ada. Each attribute is described in more detail in the following subsections.

***Table 10-3 Implementation-Defined Attributes***

| Attribute | Meaning |
|---|---|
| 'Compiler_Key | Identifies the compiler used to generate code for the specified object |
| 'Compiler_Version | Yields the version of the compiler used to generate code for the specified object |
| 'Dope_Address | Yields the address of the dope vector for an array object |

*Table 10-3 Implementation-Defined Attributes (Continued)*

| Attribute | Meaning |
|---|---|
| 'Dope_Size | Yields the size of the dope vector for an array object |
| 'Entry_Number | Uniquely identifies an entry or generic |
| 'Homogeneous | Specifies whether objects in a collection are of uniform size |
| 'Type_Key | Uniquely identifies a type |

## 'Compiler_Key

For a prefix N that denotes the name of an entity, `N'Compiler_Key` yields the full pathname of the compiler key, which indicates the compiler that was used to generate code for the unit containing the definition of N.

The entity named by N can be a program unit (package, subprogram, task, or generic), an object (variable, constant, named number, or parameter), a type or subtype (but *not* an incomplete type), or an exception.

The value returned by this attribute is of type String; for example, `"/rnv_home/keys/ada_rational_rs6k_aix"`.

This attribute can be used for runtime detection of incompatibilities in data representation. It typically is used when passing messages over a network to ensure that the reader and writer agree on how to interpret the message. See also 'Compiler_Version.

## 'Compiler_Version

For a prefix N that denotes the name of an entity, `N'Compiler_Version` yields the version of the compiler that was used to generate code for the unit containing the definition of N.

The entity named by N can be a program unit (package, subprogram, task, or generic), an object (variable, constant, named number, or parameter), a type or subtype (but *not* an incomplete type), or an exception.

The value returned by this attribute is of type string; for example, `"11.4.0"`.

This attribute can be used for runtime detection of incompatibilities in data representation. It typically is used when passing messages over a network to ensure that the reader and writer agree on how to interpret the message. See also 'Compiler_Key.

## 'Dope_Address

For an array object A, `A'Dope_Address` yields the address of the dope vector that describes A. The value is of type System.Address. If the object denoted by A has no dope vector, this value is 0.

This attribute can be used in conjunction with 'Dope_Size for retrieving information about the object, as when reconstructing the array when passing messages over a network. See "Dope Vectors" on page 76 for additional information.

## 'Dope_Size

For an array object A, `A'Dope_Size` yields the size in bits of the dope vector. The value is of type Universal_Integer.

A positive value is always returned, whether or not the object denoted by A has a dope vector. Use 'Dope_Address to determine whether the dope vector actually exists.

This attribute can be used for retrieving information about the object, as when reconstructing the array when passing messages over a network. See "Dope Vectors" on page 76 for additional information.

## 'Entry_Number

For a prefix E that denotes a task entry or generic formal subprogram, `E'Entry_Number` yields a Universal_Integer value that uniquely identifies the entity denoted by E.

## 'Homogeneous

For a prefix T that denotes an access type, `T'Homogeneous` yields a Boolean value. The value returned is True if all objects

in the collection will always have the same constraints. The converse, however, is not true.

Applying this attribute to a type that is not an access value is a semantic error.

Note that the attribute is a property of the type, not of the subtype. Thus, for any access type T, `T'Homogeneous` yields the same value as `T'Base'Homogeneous`.

For example:

```
type T1 is access String (1..10);-- T1'Homogeneous=True
type T2 is access String;         -- T2'Homogeneous=False
type T3 is new T2 (1 .. 10);      -- T3'Homogeneous=False
type T4 is new T1;                -- T4'Homogeneous=True
```

At the implementation level, the attribute indicates whether constraint information is stored with allocated objects.

## 'Type_Key

For a prefix T denoting a type name, `T'Type_Key` yields a string that uniquely identifies type T. This attribute typically is used when passing messages of a given type over a network to ensure that the reader and writer agree on the type to use when interpreting the message.

## Attributes of Numeric Types

This section lists the values returned by attributes that apply to integer types.

### Integer Types

The attributes that apply to integer types—namely, 'First, 'Last, and 'Size—yield the values shown below for the predefined base type:

***Table 10-4 Attribute Values for Integer Types***

| Attribute | Value |
|---|---|
| 'First | $-2^{31}$ |
| 'Last | $2^{31}-1$ |
| 'Size | 32 |

# Packages Standard and System

This section contains the specifications for packages Standard and System.

## Package System (LRM 13.7)

```
package System is
   type Address is private;

   type Name is (Sparc_Solaris);

   System_Name : constant Name := Sparc_Sunos;
   Storage_Unit : constant := 8;
   Memory_Size : constant := +(2 ** 31) - 1;

   Min_Int : constant := -(2 ** 31);
   Max_Int : constant := +(2 ** 31) - 1;

   Max_Digits : constant := 15;
   Max_Mantissa : constant := 31;
   Fine_Delta : constant := 1.0 / (2.0 ** 31);
   Tick : constant := 1.0 / 60.0;

   subtype Priority is Integer range 0 .. 255;

   Assertion_Error : exception;

   function To_Address (Value : Integer) return Address;
   function To_Integer (Value : Address) return Integer;

   function "+" (Left : Address; Right : Integer)
      return Address;
   function "+" (Left : Integer; Right : Address)
      return Address;
   function "-" (Left : Address; Right : Address)
      return Integer;
   function "-" (Left : Address; Right : Integer)
      return Address;

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

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

   -- The functions above are unsigned in nature. Neither
   -- Numeric_Error nor Constraint_Error will ever be
   -- propagated by these functions.
   --
   -- Consequently,
   --
   --    To_Address (Integer'First) >
   --       To_Address (Integer'Last);
   --
   -- and
   --
   --    To_Address (0) < To_Address (-1);
   --
   -- The unsigned range of Address includes values that
   -- are larger than those implied by Memory_Size.

   Address_Zero : constant Address;
   Null_Address : constant Address;
   No_Addr : constant Address;

private

   type Address is new Integer;

   Address_Zero : constant Address := 0;
   Null_Address : constant Address := 0;
   No_Addr : constant Address := 0;

   pragma Suppress(Elaboration_Check, On => System."+");
   pragma Suppress(Elaboration_Check, On => System."-");
   pragma Suppress(Elaboration_Check, On => System.">");
   pragma Suppress(Elaboration_Check, On => System.">=");
   pragma Suppress(Elaboration_Check, On => System."<");
   pragma Suppress(Elaboration_Check, On => System."<=");
   pragma Suppress(Elaboration_Check,
      On => System.To_Address);
   pragma Suppress(Elaboration_Check,
      On => System.To_Integer);

   pragma Inline(System."+");
   pragma Inline(System."-");
   pragma Inline(System.">");
   pragma Inline(System.">=");
   pragma Inline(System."<");
```

```
   pragma Inline(System."<=");
   pragma Inline(System.To_Address);
   pragma Inline(System.To_Integer);

end System;
```

## Package Standard (LRM Annex C)

```
package Standard is
   type *Universal_Integer* is ...
   type *Universal_Real* is ...
   type *Universal_Fixed* is ...
   type Boolean is (False, True);
   type Integer is range -2147483648 .. 2147483647;
   type Float is digits 6
      range -((2.0 ** 128) - (2.0 ** 104)) ..
      ((2.0 ** 128) - (2.0 ** 104));-- about   3.4E+38
   type Long_Float is digits 15
      range -((2.0 ** 1024) - (2.0 ** 971)) ..
      ((2.0 ** 1024) - (2.0 ** 971)); -- about   1.8E+308
   subtype Natural is Integer range 0 .. 2147483647;
   subtype Positive is Integer range 1 .. 2147483647;
   type Duration is delta 0.000061035156
      range -131072.00000000 ..
         0131071.999938964843750000000;
   type Character is ...

   package Ascii is...

   type String is array (Positive range <>) of Character;
   Constraint_Error : exception;
   Numeric_Error : exception;
   Storage_Error : exception;
   Tasking_Error : exception;
   Program_Error : exception;
   type *Anytype* is
      record
         null;
      end record;
end Standard;
```

The following table shows the sizes of predefined integer and floating-point types:

*Table 10-5 Sizes of Predefined Numeric Types*

| Ada Type Name | Size |
|---|---|
| Integer | 32 bits |
| Float | 32 bits |
| Long_Float | 64 bits |

Fixed-point types are implemented using 32 bits.

Floating-point types are implemented according to the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985).

Standard.Duration is a 32-bit fixed-point type with a delta of $2^{-14}$.

# Representation Clauses

This section discusses limitations on representation clauses in the following categories:



For related information, see Chapter 9, "Sizes of Objects."

## Representation-Clause Error Handling

Normally, an invalid representation clause causes an error at compile time and prevents successful compilation.

Several Apex context switches, however, allow you to specify whether to treat certain classes of invalid representation clauses as nonfatal errors that allow successful compilation rather than as errors. See Appendix A, "Switches," and *Using Rational Apex* for details on the following switches:

- Ignore_Invalid_Rep_Specs: Affects the handling of both invalid and unsupported representation specifications.
- Ignore_Unsupported_Rep_Specs: Affects the handling of unsupported representation specifications only.

## Length Clauses

Length clauses are never allowed on derived record types; otherwise, length clauses are supported by Rational Ada as follows:

- The value of a 'Size attribute must be a positive static integer expression. It must be greater than or equal to the minimum size necessary to store the largest possible value of the type. 'Size attributes are supported for all scalar and composite types with the following restrictions:

***Table 10-6 'Size Attribute Restrictions***

| Types | Legal Attribute Values |
|---|---|
| Access and task | 32 |
| Composite | Must not imply compression of composite components; such compression must have been explicitly requested using a length clause or pragma Pack on the component type |
| Discrete | Less than or equal to 32 |
| Fixed-point | Less than or equal to 32 |
| Floating-point | Can specify only the size the type would have if there were no clause; therefore, the only legal values are 32 and 64 |

- 'Storage_Size attributes are supported for access and task types. The value given by a 'Storage_Size attribute can be any integer expression, and it is not required to be static.
- 'Small attributes are supported for fixed-point types. The value given by a 'Small attribute must be a positive static real number that cannot be greater than the delta of the base type. It need not be a power of 2.

### Enumeration Representation Clauses (LRM 13.3)

Enumeration representation clauses are supported with the following restriction:

- The allowable values for an enumeration clause range from IntegerFirst to IntegerLast.

### Record Representation Clauses (LRM 13.4)

Both full and partial representation clauses are supported for both discriminated and undiscriminated records. Record component clauses are not allowed on:

- Array or record fields whose constraint involves a discriminant of the enclosing record
- Array or record fields whose constraint is not static

The static simple expression in the alignment clause part of a record representation clause—see the Ada LRM 13.4 (4)—must be a power of 2 with the following limits:

```
1 <= static_simple_expression <= 16
```

The size specified for a discrete field in a component clause must not exceed 32 bits.

### Change of Representation (LRM 13.6)

Change of representation is supported wherever it is implied by support for representation specifications. In particular, type conversions between array types may cause packing or unpacking to occur; conversions between related enumeration types with different representations may result in table-lookup operations.

## Implementation-Generated Names

The Ada LRM allows for the generation of names denoting implementation-dependent components in records. No such names are visible to the user for Rational Ada.

# Address Clauses (LRM 13.5)

Address clauses cannot be applied to task types. No other restrictions are placed on address clauses.

An address clause can be attached to a task entry only when the task entry is used for signal (interrupt) catching; however, in this case, the task entry *must* be available at the time of the signal. See the discussion of pragma Signal_Handler on page 99 and "Interrupt-Entry Tasks" on page 40 for additional information.

Values of address clauses are not checked for validity. No check is made to determine whether an address clause causes the overlay of objects or of program units.

# Unchecked Programming

## Unchecked Storage Deallocation (LRM 13.10.1)

Unchecked storage deallocation is implemented by the Ada LRM-defined generic function Unchecked_Deallocation. This procedure can be instantiated with an object type and its access type, resulting in a procedure that deallocates the object's storage. Objects of any type can be deallocated.

The storage reserved for the entire collection associated with an access type is reclaimed when the program exits the scope in which the access type is declared. Placing an access-type declaration within a block can be a useful implementation strategy when conservation of memory is necessary within a collection. Space on the free list is coalesced when objects are deallocated.

Erroneous use of dangling references may be detected in certain cases. When detected, the Storage_Error exception is raised. Deallocation of objects that were not created through allocation (that is, through Unchecked_Conversion) may also be detected in certain cases, also raising Storage_Error.

## Unchecked Type Conversion (LRM 13.10.2)

Unchecked type conversion is implemented by the generic function Unchecked_Conversion defined by the Ada LRM. This

function can be instantiated with *source* and *target* types, resulting in a function that converts source data values into target data values.

Unchecked type conversion moves storage units from the source object to the target object sequentially, starting with the lowest address. Transfer continues until the source object is exhausted or the target object runs out of space. If the target is larger than the source, the remaining bits are undefined. Depending on the target-computer architect, the result of conversions may be right- or left-justified.

### Restrictions on Unchecked Type Conversion

The following restrictions apply to unchecked type conversion:

- The target type of an unchecked type conversion cannot be an unconstrained array type or an unconstrained discriminated type without default discriminants.
- Internal consistency among components of the target type is not guaranteed. Discriminant components may contain illegal values or be inconsistent with the use of those discriminants elsewhere in the type representation.

## Input/Output Packages

The Ada language defines specifications for four I/O packages: Sequential_Io, Direct_Io, Low_Level_Io, and Text_Io. The following subsections explain the implementation-dependent characteristics of those four packages provided with Rational Ada.

### Sequential_Io (LRM 14.2.2 and 14.2.3)

For the Read procedure of Sequential_Io, the Data_Error exception is raised only when the size of the data read from the file is greater than the size of the *out* parameter item.

### POSIX Compliance

The Form parameter on subprograms in Sequential_Io is compliant with the POSIX.5 standard to the extent described in Chapter 7, "Files and I/O."

## Direct_Io (LRM 14.2.4)

Package Direct_Io may not be instantiated with any type that is either an unconstrained array type or a discriminated record type without default discriminants. A semantic error is reported when an attempt is made to install any unit that contains an instantiation in which the actual type is such a forbidden type.

For the Read procedure of Direct_Io, no check is performed to ensure that the data read from the file can be interpreted as a value of the Element_Type.

### Specification of Package Direct_Io (LRM 14.2.5)

The declaration of the type Count in package Direct_Io is:

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

where Element_Type is the generic formal type parameter.

### POSIX Compliance

The Form parameter on subprograms in Direct_Io is compliant with the POSIX.5 standard to the extent described in Chapter 7, "Files and I/O."

## Low_Level_Io (LRM 14.6)

Package Low_Level_Io is not provided with Rational Ada.

## Text_Io (LRM 14.3)

The Text_Io default input and output files are associated with the UNIX standard input and standard output paths, respectively.

### Specification of Package Text_Io (LRM 14.3.10)

The declaration of the type Count in Text_Io is:

```
type Count is new integer range 0 .. 1_000_000_000;
```

The declaration of the subtype Field in Text_Io is:

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

### File-Management Operations

The operations of Get and Put are as described in the Ada LRM.

Data written using Put and Put_Line is not interpreted in any fashion. Data written using Put_Line is followed by the line terminator Ascii.Lf.

Data read using Get and Get_Line is not interpreted except that the line terminator, Ascii.Lf, and the page terminator, Ascii.Ff, are removed from the input stream.

### POSIX Compliance

The Form parameter on subprograms in Text_Io is compliant with the POSIX.5 standard to the extent described in Chapter 7, "Files and I/O."

## Other Implementation-Dependent Features

### Machine Code (LRM 13.8)

Machine-code insertions are not supported at this time.